General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

MCR 75-390
NAS8-31616

Volume II

Phase II
Final
Report

NASA CR-
144529

October 1975

Introduction to
PLANS Programming

Scheduling Language and Algorithm Development Study

MARTIN MARIETTA

FOREWORD

This is the final report for Phase II of the Scheduling Language and Algorithm Development Study (NAS9-13616). It is contained in three volumes. The objectives of Phase II were to implement prototypes of the Scheduling Language called PLANS and the scheduling module library that were designed and specified in Phase I.

Volume I of this report contains data and analyses related to a variety of algorithms for solving typical large-scale scheduling and resource allocation problems. The capabilities and deficiencies of various alternative problem solving strategies are discussed from the viewpoint of computer system design.

Volume II is an introduction to the use of the Programming Language for Allocation and Network Scheduling (PLANS). It is intended as a reference for the PLANS programmer.

Volume III contains the detailed specifications of the scheduling module library as implemented in Phase II. This volume extends the Detailed Design Specifications previously published in the Phase II Interim report (April 1975).

TABLE OF CONTENTS

FIGURES

1.0 INTRODUCTION

The design of the Programming Language for Allocation and Network Scheduling (PLANS) was prompted by the inadequacies of existing languages being used to solve scheduling problems. A high-level language was needed that would allow easy, direct expression of the kinds of functions frequently found in scheduling and resource allocation programs. PLANS fulfills this need primarily because of its unique capability to allow dynamic manipulation of tree data structures at execution time. Another important feature is the close correspondence that exists between basic scheduling functional operations and PLANS statements. This allows both the initial programmer and the maintenance programmer to easily design and modify PLANS programs. These powerful language features make it applicable to many areas other than scheduling. That is, PLANS is not a special purpose scheduling language, even though it was motivated by scheduling problems. It is a generalized, high-level tree manipulation language.

1.1 PURPOSE OF THE DOCUMENT

This User Guide is intended to provide sufficient information about PLANS to allow the reader who has some computer programming experience to construct correct and useful PLANS programs, using the entire set of functional capabilities which PLANS currently provides. The basic philosophy of PLANS will be discussed first, to provide some intuitive feeling for the nature and unique properties of the language. PLANS access and update reference techniques will then be described, providing the background information necessary for the presentation of each of the basic statements of

PLANS. These statements will be discussed in some detail, with examples illustrating the major variants of the statements. Several complete programs will then be presented and discussed in sufficient detail to provide a working knowledge of some of the techniques of problem solution to which PLANS lends itself.

1.2 BACKGROUND OF PLANS

Although its capabilities have proven to be much more broadly applicable, PLANS was designed to achieve a single goal: to allow the designer of experimental, constantly changing scheduling and resource allocation algorithms to translate his algorithm designs to working code directly from their basic functional descriptions, without intermediate detailed program design steps, without highly specialized programming expertise, and at minimum span time and manpower costs. The necessity to go through several additional design and implementation steps before the advent of PLANS resulted in unacceptably long development times and high costs. Equally important, it tended to discourage the truly experimental approach to scheduling algorithm development which holds the greatest promise of convergence on good solutions for large, logically complex scheduling problems. PLANS was designed, then, to cut development cost and span time, and also to provide a medium for easy modification of scheduling programs.

An analysis of previously existing programming languages as applied to scheduling problems revealed two deficiencies: (1) the language level did not correspond to the level of the functions typical in scheduling algorithms, and (2) more significantly, the

data structures (usually only arrays) of the languages did not correspond to those typical of scheduling problems.

Scheduling problems typically involve information structures which are logically hierarchical. A schedule consists of jobs, each of which has certain properties of its own (time of occurrence, duration, name, etc.), and each of which also has certain relations to other jobs (predecessors, etc.) and to particular resources which are required to perform the jobs. These resource assignments have, in turn, such properties as time of occurrence, duration, etc. The inputs to scheduling algorithms are also typically hierarchic in nature, involving, for example, information about resources, which breaks down into resource types, each of which in turn may involve many resource units, each of which has its own physical and logical properties (weight, location, etc.), and each of which is also unavailable at certain times due to prior assignments to jobs. The necessity to represent information of this sort in the form of arrays (as when programming in FORTRAN, for example) led to programs which were quite large, difficult, and unreadable. This is due to the overwhelming preponderance of indexing operations and similar functions required to express, in array form, information which is not logically of an array character.

As a result of these considerations, PLANS was designed around a single feature which is unique among high-level languages: the provision of hierarchic data structures -- trees -- whose structure, as well as data content, can be manipulated at execution time. Many languages (e.g., COBOL, PL/1, ALGOL) have hierarchic

data structures which are static during execution. The feature of PLANS which is novel (except, perhaps, among difficult-to-use list-processing languages), is its dynamic manipulation of trees. The output usually required of scheduling programs is, in large part, a restructuring of the input, which can be most easily accomplished in a language which allows direct restructuring of its data structures.

1.3 BASIC DESIGN PHILOSOPHY

Because it is intended to be used by problem area experts, rather than programming experts, the language has been designed to minimize functionally nonessential details, such as data type declarations, entry declarations, etc. Such language features usually allow the programmer greater control over the detailed execution of his programs, but require greater programmer sophistication and more difficult program logic. In any case, such features are more appropriate in languages which are intended to handle quantitative problems. While PLANS provides quantitative capabilities, its emphasis is more on manipulation of data structures, which has proven to be the principal activity performed in most scheduling algorithms. Although future extensions of PLANS may very well allow the more sophisticated programmer to use type declarations, etc., it is basic to the philosophy of PLANS that such program features should never be unnecessarily required.

PLANS data access and update capabilities have been made sufficiently powerful to allow PLANS statements to correspond nearly one-to-one with the functional elements of typical algorithm

specifications. Dynamic tree manipulation, one of the features which helps accomplish this, has already been discussed. PLANS also provides data access capabilities which allow some operations which are logically iterative to be performed in a single statement or substatement. Basically, PLANS is oriented toward qualificational, rather than conditional, statements. Conditional construction, which is much more common, is also more difficult and less natural to use. A simple conditional construct might take the form, "Look at each box in turn. If the box you are looking at is red, then cease looking and go to a special address, at which you will be instructed to pick up the box you have found." Although constructs of this sort are possible in PLANS, the language is more oriented toward qualificational constructs, such as "Pick up the box which is red." This orientation results in a considerable increase in the power of the individual statement.

Throughout the language, emphasis has been placed on simplicity and generality of function. Even though the basic design goal of the language is to allow more effective programming of scheduling and resource allocation algorithms, the language contains no specialized functions like "Schedule", "Unschedule", etc. Such specialized functions could only exist if they could be precisely defined in a way that is invariant across different problem types and programmers. Such specialized language capabilities often serve more to constrain the programmer than to aid him. While PLANS does provide a few somewhat specialized functions (DO FOR ALL COMBINATIONS, DO FOR ALL PERMUTATIONS, ORDER), its emphasis is really on appropriate data structures, access methods, and manipulative operations.

2.0 THE BASIC STATEMENT SET

2.1 PLANS TREES

The principal difference between PLANS and other programming languages is that PLANS is oriented primarily toward the manipulation of ordered, labeled tree structures. This section will describe these structures, illustrating their properties in graphical form, and the next section will acquaint the reader with the various mechanisms available for accessing information in PLANS trees.

The tree is a hierarchic data structure which consists of a number of nodes. Figure 2-1 illustrates a typical tree: the root node is shown at the top and other nodes branch out beneath it. The tree has a name ("$PAYLOAD"), which starts with a dollar sign to allow the translator to distinguish it from variable names, key words, etc. Whenever it occurs in a program, the tree name is a reference to the root node and its entire substructure, if any. Figure 2-2 illustrates a tree which has no substructure.

Each node has a label. For consistency in format, we will always write the label of a node to the right of the node. A label can consist of any character string (of length 30 or less) containing no blanks, or the label can be null. A null label is indicated by a special character, the cent sign ("¢"), although, for convenience, the label is sometimes omitted altogether when trees are displayed graphically. In Figure 2-1, the only node shown with a null label is the root node, but any node can conceivably have a null label. Note that the name of the tree is not the same as the label of the root node, although they may be identical.

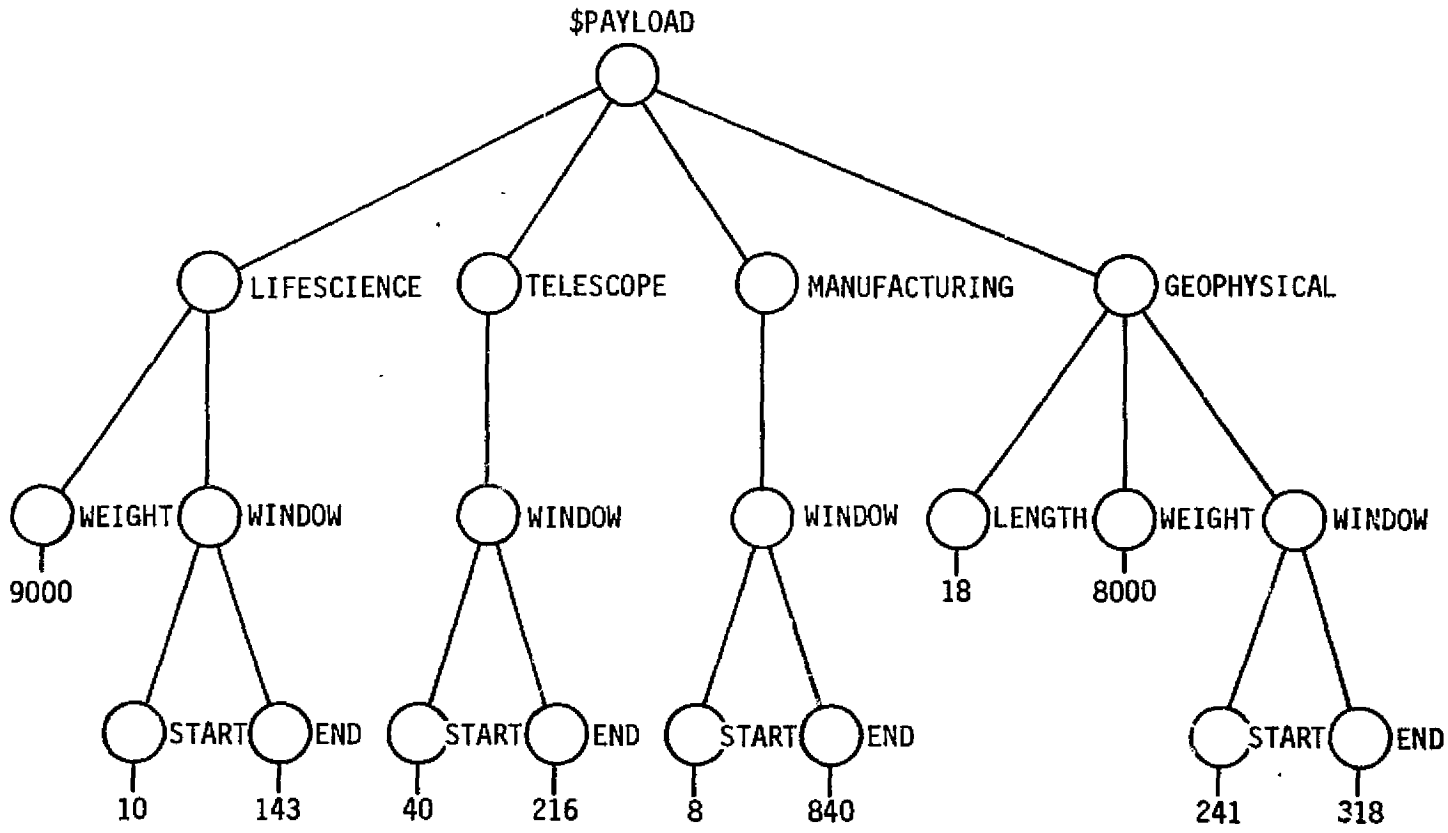


Fig. 2-1 A Labeled Tree (with substructure)

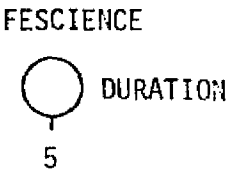


Fig. 2-2 A Labeled Tree (with value)

Labels are useful in two ways: they can be used simply to store information in the tree, usually to identify the nature of the information in a subtree, but their more important property is that they provide a means of directly accessing information. It is not necessary to search all the way through the information about a payload, for example, in order to determine its weight. The weight can be accessed directly. If information is to be accessed directly by label in a PLANS program, however, it is necessary that the label character string satisfy some more rigid constraints than those given above. It can still consist of up to 30 characters, but the first character must be alphabetic and all others must be either alphanumeric or the special underbar ("_") character. The underbar character allows one to use meaningful and readable labels (e.g., THIS_IS_A_READABLE_NAME).

The nodes exactly one level below a given node are called its descendants, or subnodes. The root node of the tree in Figure 2-1 has four descendants. The node labeled LIFESCIENCE has two descendants, which, in turn, are labeled WEIGHT and WINDOW. Nodes that have descendants are called nonterminal nodes; nodes without descendants are called terminal nodes. There are 11 terminal and 9 nonterminal nodes in the figure.

The node exactly one level above a given node is called its ascendant. The root node of the tree in figure 2-1 is the ascendant of the nodes labeled LIFESCIENCE, TELESCOPE, MANUFACTURING and GEOPHYSICAL. The node labeled GEOPHYSICAL is, in turn, the ascendant of the node labeled LENGTH. Note that any node can have, at most, one ascendant.

In addition to a label, a terminal node has a value, which may be a character string, a numeric value or null. Values are shown below their nodes. Thus, the node at the bottom right of Figure 2-1 has the label "END" and the value "318". Labels and values are character strings which may, depending on the context, have numerical significance. Like labels, values may consist of up to 30 characters with no embedded blanks.

While the graphical format is convenient for displaying conceptual tree structures and for demonstrating the effect of specific PLANS statements, it is too cumbersome and rigid for convenient use in the display of specific tree structures, especially large ones. For this purpose, the indented text format is used. The tree of Fig. 2-1 is expressed in the form shown in Fig. 2-3. In this case, the structure is defined by the indentation pattern, rather than by node-connecting lines. Each line of text represents a node. The information occurring first on a line is the node label, while the values of terminal nodes are separated from the corresponding labels by a hyphen (-) character that is surrounded by blanks. In order to allow rigorous definition of tree structures in which some nodes have null labels, it is necessary to employ a special convention for representing them. Null labels are represented by a cent sign (¢). This convention is occasionally employed in the graphical format, although it is unnecessary there.

```
$PAYLOAD
     LIFESCIENCE
          WEIGHT - 9000
          WINDOW
               START - 10
               END - 143
     TELESCOPE
          WINDOW
               START - 40
               END - 216
     MANUFACTURING
          WINDOW
               START - 8
               END - 840
     GEOPHYSICAL
          LENGTH - 18
          WEIGHT - 8000
          WINDOW
               START - 241
               END - 318
```

Figure 2-3 The Tree of Fig. 2-1 in Textual Format

An additional convention, which has been adopted, is the use of parenthesized labels and values to represent variable data in the definition of a particular tree application. If a label or value occurs without parentheses, it is assumed that the character string shown is literally present in the tree. For example, the tree

```
$PAYLOAD
     LIFESCIENCE
          WEIGHT - 9000
          LENGTH - 27
```

contains only actual values and labels. But if one wanted to show only the nature of the information contained in this tree, the following form might be used.

```
$PAYLOAD
     (PAYLOAD NAME)
          (CHARACTERISTIC) - (VALUE)
          (CHARACTERISTIC) - (VALUE)
               .
               .
               .
     (PAYLOAD NAME)
          .
          .
          .
```

2.2 PLANS Tree References

PLANS provides the programmer with a number of simple and powerful means of accessing and updating the information contained in the labelled tree structures discussed in the previous section. These methods are based on the notion that the programmer can "point" to a particular tree node by specifying which tree it is in, which descendant of the root node it is under, which descendant of that node it is under, etc. (Remember that the term "descendant," as used here, means immediate descendant.)

This section will describe these methods of the referencing, illustrating their properties with simple examples, and the next section will acquaint the reader with the various PLANS statements which utilize these tree reference methods.

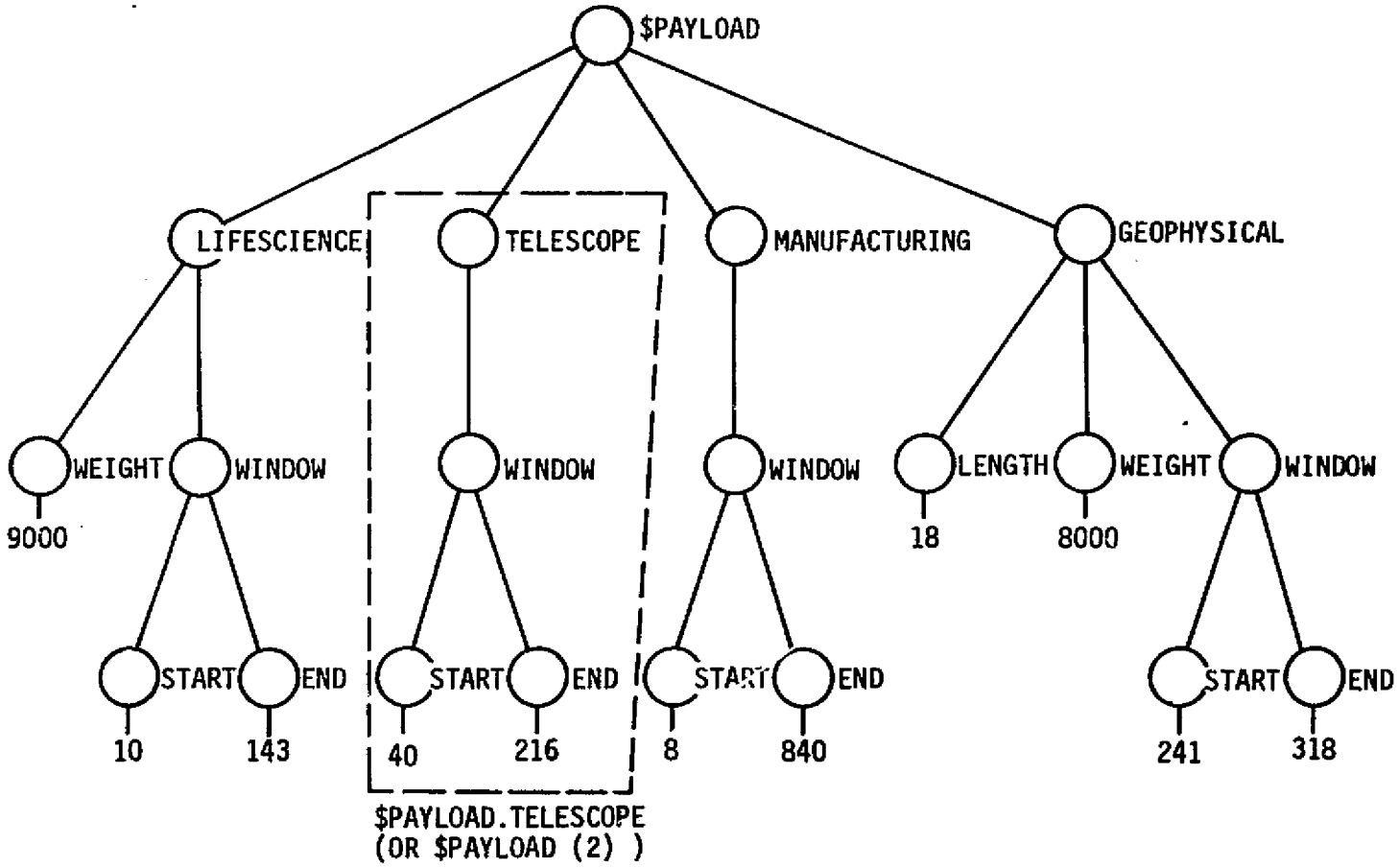
2.2.1 Tree Reference By Label

There are two basic ways of specifying which subnode of a given node is the relevant one. The first way of specifying a subnode is by its label.

Suppose, for example, that information about the telescope payload in figure 2-4 is desired. Because the name of the tree is $PAYLOAD and the name of the payload in question is TELESCOPE, the programmer might write $PAYLOAD.TELESCOPE to access this information. This is an example of qualification by label.

Access qualified by label can be continued to any desired depth in the tree. Consider, for example, the node with value 216 in figure 2-4. $PAYLOAD.TELESCOPE.WINDOW.END is one way of pointing to this node.

Fig. 2-4 Basic Tree Access Mechanisms



Certain words, because they are subscript keywords (see following sections), cannot be used as labels. They are "FIRST," "LAST", "NEXT" and "ALL". Thus, $X.NEXT is now allowed.

The function keywords "LABEL" and "NUMBER" (see Sections 2.2.9, 2.2.10) also cannot be used as labels.

2.2.2 Tree Reference By Ordinal Position (Subscript)

The second basic way of specifying subnodes is by ordinal position, or subscript. As was mentioned before, but not explained in detail, PLANS trees are ordered trees; that is, the ordering of the descendants of a node is significant. Unless action is taken to change or reorder a tree structure, the order remains constant. In the previous example (see figure 2-4), since TELESCOPE is the second payload, information about that payload can be referred to as $PAYLOAD(2), as well as $PAYLOAD.TELESCOPE. The usefulness of referencing a node by subscript becomes apparent when the node to be referenced has no label.

Access qualified by subscript can be continued to any desired depth in the tree. Consider, for example, the node with value 216 in figure 2-4. One way of pointing to this node is $PAYLOAD(2)(1)(2).

Labels and subscripts can be mixed at will when specifying a tree node. For example, several ways of specifying the node with value 216 in figure 2-4 are:

```
$PAYLOAD.TELESCOPE.WINDOW(2)
$PAYLOAD.TELESCOPE(1).END
$PAYLOAD(2).WINDOW(2)
```

"FIRST", "LAST", "NEXT" and "ALL" are keywords which may be used as subscripts. Their use will be explained in detail in the following sections. Note that "FIRST", "LAST", "NEXT" and "ALL" may not appear as labels (i.e. $X.NEXT is not allowed).

2.2.3 Tree Reference By Subscript Keyword ("FIRST")

The first subnode at a given level can be referenced using the subscript keyword "FIRST", as in $PAYLOAD(FIRST),which is illustrated in figure 2-5. $PAYLOAD(FIRST) is equivalent to the subscript specification $PAYLOAD(1), but is executed more efficiently.

Since labels and subscripts can be mixed at will when specifying a tree node, the following are several valid ways of pointing to the node with value 216 in figure 2-5:

```
$PAYLOAD(2)(FIRST)(2)
$PAYLOAD.TELESCOPE(FIRST).END
$PAYLOAD.TELESCOPE(FIRST)(2)
$PAYLOAD(2)(FIRST).END
```

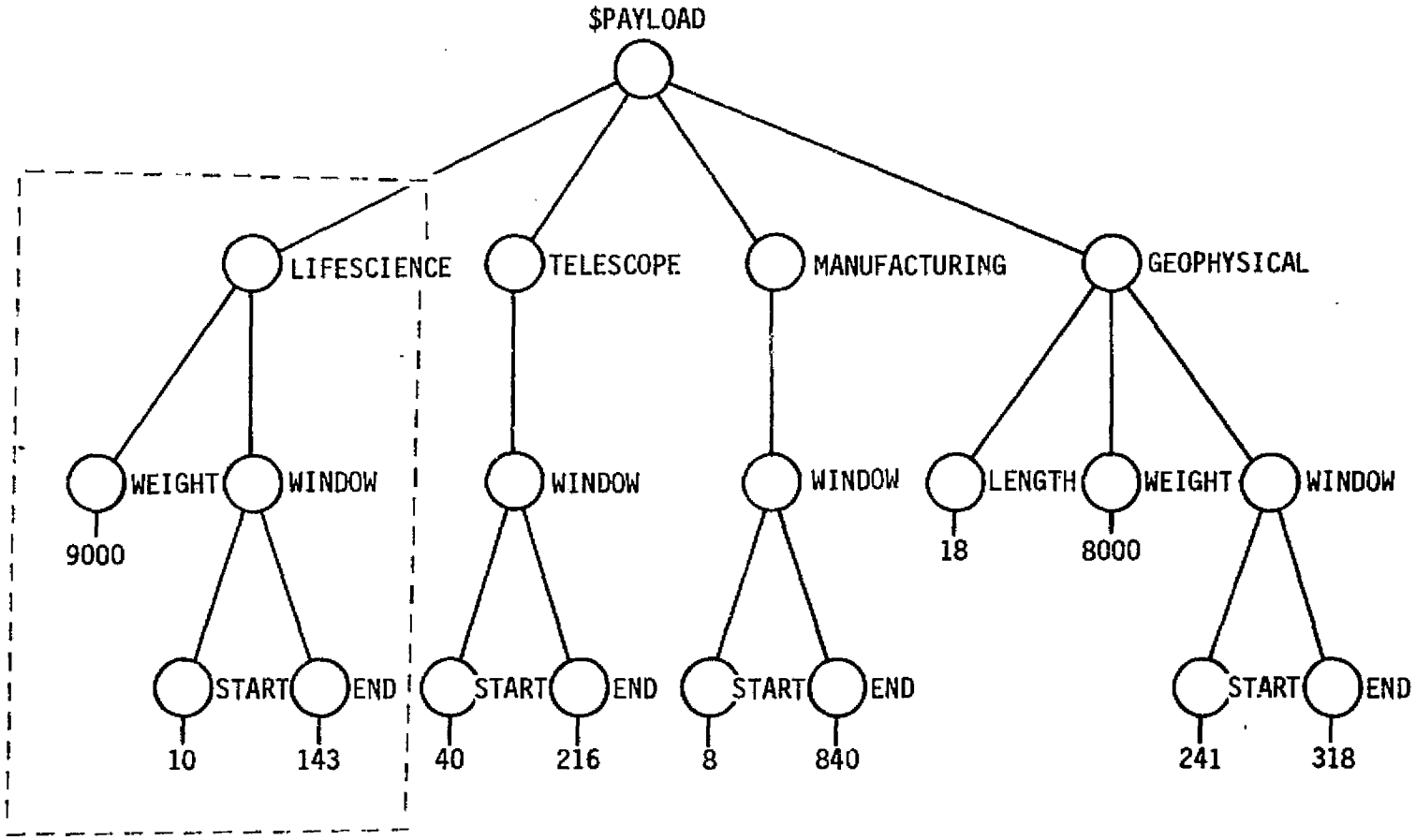
Note that the subscript keyword "FIRST" may not appear as a tree node label (i.e. $X.FIRST is not allowed).

2.2.4 Tree Reference By Subscript Keyword ("LAST")

The last subnode at a given level can be referenced using the subscript keyword "LAST", as in $PAYLOAD(LAST), which is illustrated in figure 2-6. Note that in figure 2-6, $PAYLOAD(LAST) is equivalent to $PAYLOAD(4), and to $PAYLOAD.GEOPHYSICAL.

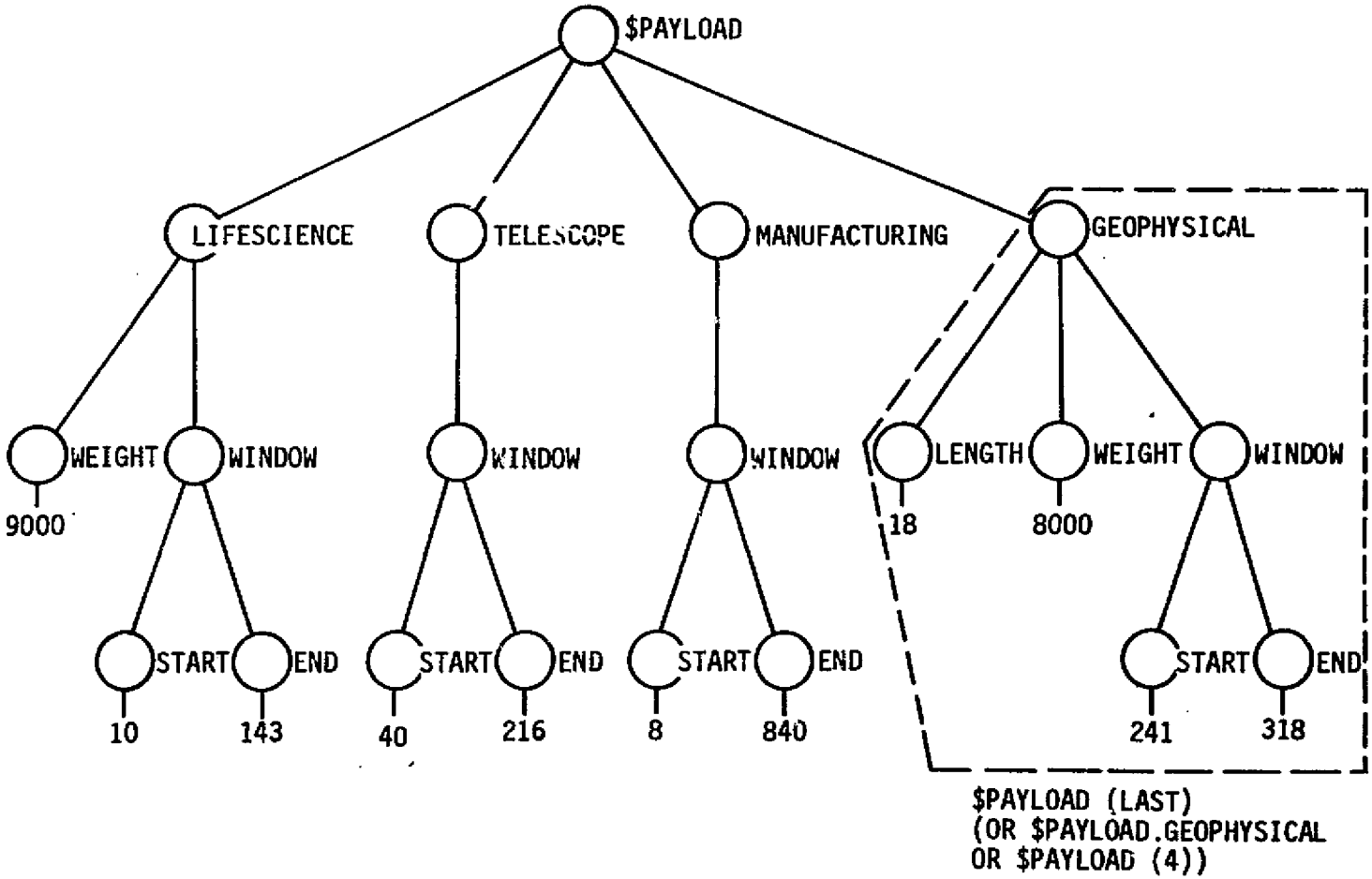
Since labels and subscripts can be mixed at will when specifying tree nodes, the following are several valid ways of pointing to the node with value 216 in figure 2-6:

Fig. 2-5 Tree Access By Subscript Keyword ("FIRST")




```
$PAYLOAD(FIRST)
(OR $PAYLOAD.LIFESCIENCE
 OR $PAYLOAD(1))
```

Fig. 2-6 Tree Access By Subscript Keyword ("LAST")



$PAYLOAD(2)(LAST)(LAST)

$PAYLOAD.TELESCOPE(FIRST)(LAST)

$PAYLOAD(2)(LAST).END

$PAYLOAD.TELESCOPE(LAST).END

$PAYLOAD.TELESCOPE(LAST)(2)

Note that the subscript keyword "LAST" may not appear as a tree node label (i.e. $X.LAST is now allowed).

2.2.5 Tree Reference By Subscript Keyword ("NEXT")

In appropriate contexts, a new subnode can be established at the right by using the subscript keyword "NEXT". For example, $PAYLOAD(NEXT) causes a new subnode to be created to the right of $PAYLOAD(4) in figure 2-7.

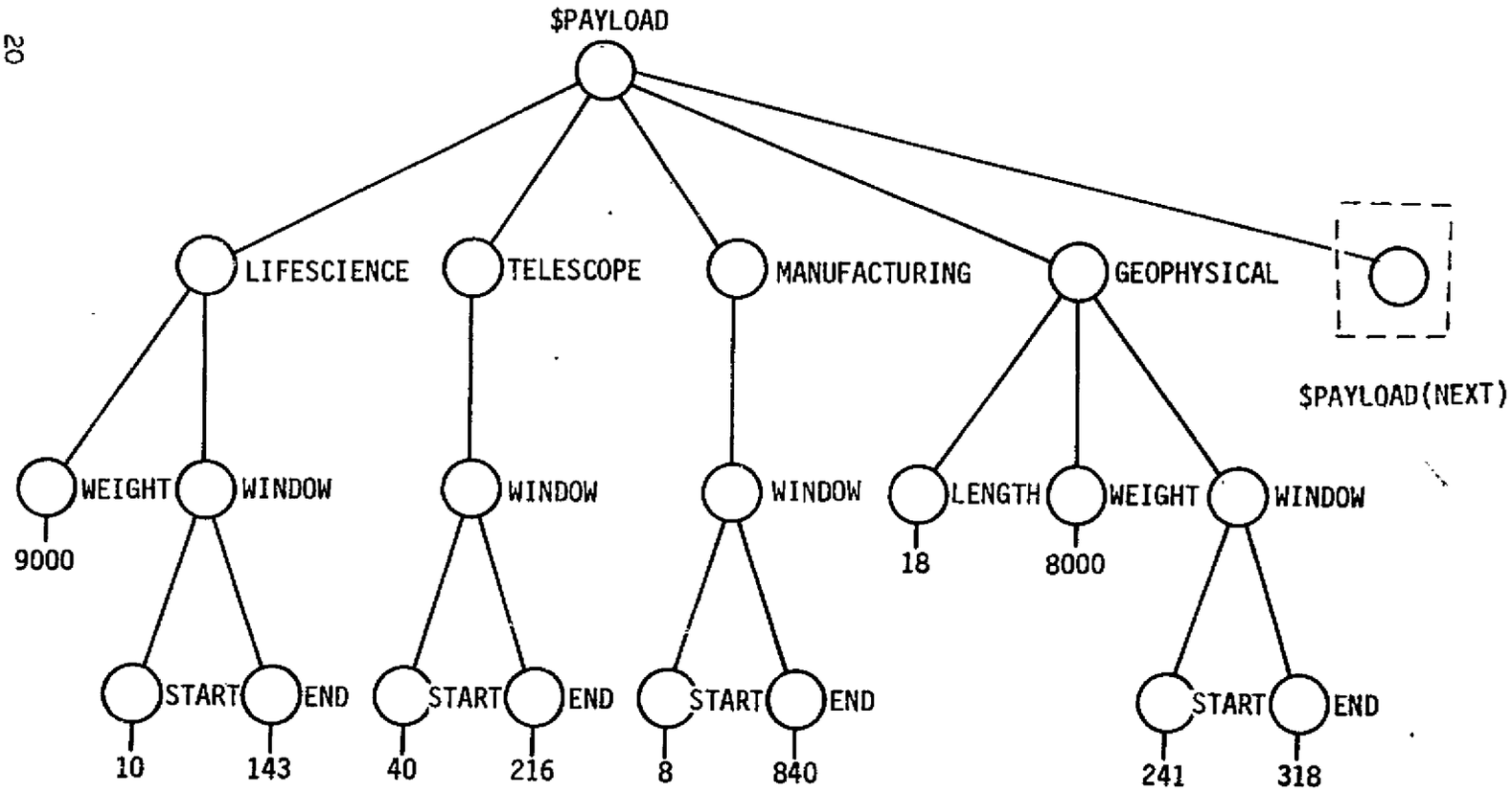
"NEXT" is meaningful only in the context of updates, for example in the tree assignment statement (see section 3.1.1) $X(NEXT) = 5 which causes a new node to be created in the tree $X which has a value of 5. "NEXT" is mentioned here in this discussion of tree references because of its similarity to "LAST".

Note that "NEXT" may not appear as a tree node label (i.e. $X.NEXT is not allowed).

2.2.6 Conditional Tree Reference By Subscript Keyword ("FIRST:")

PLANS provides conditional access mechanisms which allow access to one or more subnodes which satisfy a specified condition. The conditional access mechanisms are the keywords "FIRST:condition" and "ALL:condition". "ALL:condition" will be discussed in the following section.

Fig. 2-7 Tree Update Mechanism ("NEXT")



The "FIRST:" ("first such that...") conditional access constitutes a reference to the first (that is, leftmost) subnode which satisfies a particular condition, where the condition is a Boolean expression. Figure 2-8 illustrates a specific example. Here, a programmer wishes to refer to the first payload whose launch window is at least 150 days long, so he writes:

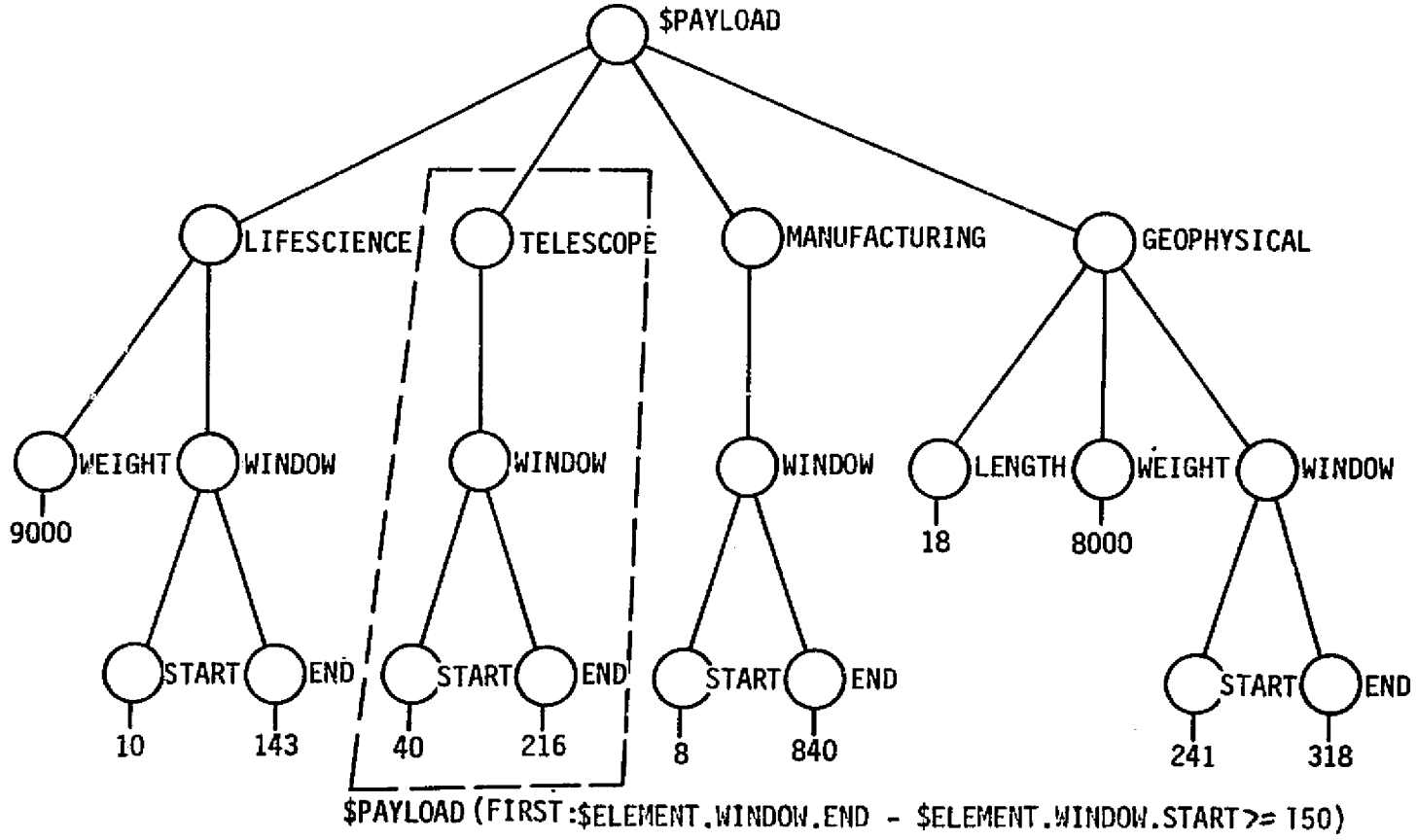
```
$PAYLOAD(FIRST:$ELEMENT.WINDOW.END - $ELEMENT.WINDOW.START >= 150).
```

The string "FIRST:" might be read "first subnode such that..." the specified condition is satisfied. $ELEMENT is a tree pointer (see section 3.2) in this context, and represents a reference to the particular subnode being considered at the current instant in the left-to-right search. Expressed in procedural terms, the operation might be, "going from left to right, consider each element (i.e. each descendant of $PAYLOAD) in turn. Calculate the launch window duration of the element being considered. If greater than or equal to 150, proceed as if it had been referenced by subscript (since "FIRST:" is a subscript keyword)." If no subnode can be found to satisfy the condition, $PAYLOAD(FIRST:$ELEMENT.WINDOW.END - $ELEMENT.WINDOW.START >= 150) will refer to a null node (see section 2.2.11), a node with no label and no value or substructure.

The conditional qualification can be combined with other qualification methods. Thus, if the programmer wished to refer to the start time of the launch window for the first payload whose launch window is at least 150 days long, he might write:

```
$PAYLOAD(FIRST:$ELEMENT.WINDOW.END - $ELEMENT.WINDOW.START >= 150).
WINDOW(FIRST) or, equivalently,
$PAYLOAD(FIRST:$ELEMENT.WINDOW.END-$ELEMENT.WINDOW.START >= 150)
(1).START
```

Fig. 2-8 Conditional Access Using Qualifier "FIRST:"



$PAYLOAD(FIRST:$ELEMENT.WINDOW.END-$ELEMENT.WINDOW.START >= 150).WINDOW.START

$PAYLOAD(FIRST:$ELEMENT.WINDOW.END-$ELEMENT.WINDOW.START >= 150)(1)(1)

$PAYLOAD(FIRST:$ELEMENT.WINDOW.END-$ELEMENT.WINDOW.START >= 150)(LAST)(1).

2.2.7 Conditional Tree Reference By Subscript Keyword ("ALL:")

"ALL:condition" is the second of two conditional access mechanisms. The "ALL:" ("all such that...") conditional access refers to all the subnodes which satisfy a particular condition, where the condition is a Boolean expression.

Figure 2-9 illustrates a specific example. Here, a programmer wishes to refer to all payloads with launch windows at least 150 days long, so he writes:

$PAYLOAD(ALL:$ELEMENT.WINDOW.END - $ELEMENT.WINDOW.START >= 150).

The string "ALL:" might be read "all subnodes such that..." the specified condition is satisfied. $ELEMENT is a tree pointer (see section 3.2) in this context, and represents a reference to the particular subnode being considered at the current instant in the left-to-right search.

Basically, the ALL: access is a reference to all the subnodes which satisfy the stated condition as shown in Fig. 2-9. The exact meaning of this type of access is somewhat context-specific, however. For a discussion of the way in which this access is interpreted in the various statements in which it can occur, see the sections dealing with those statements (sections 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.5).

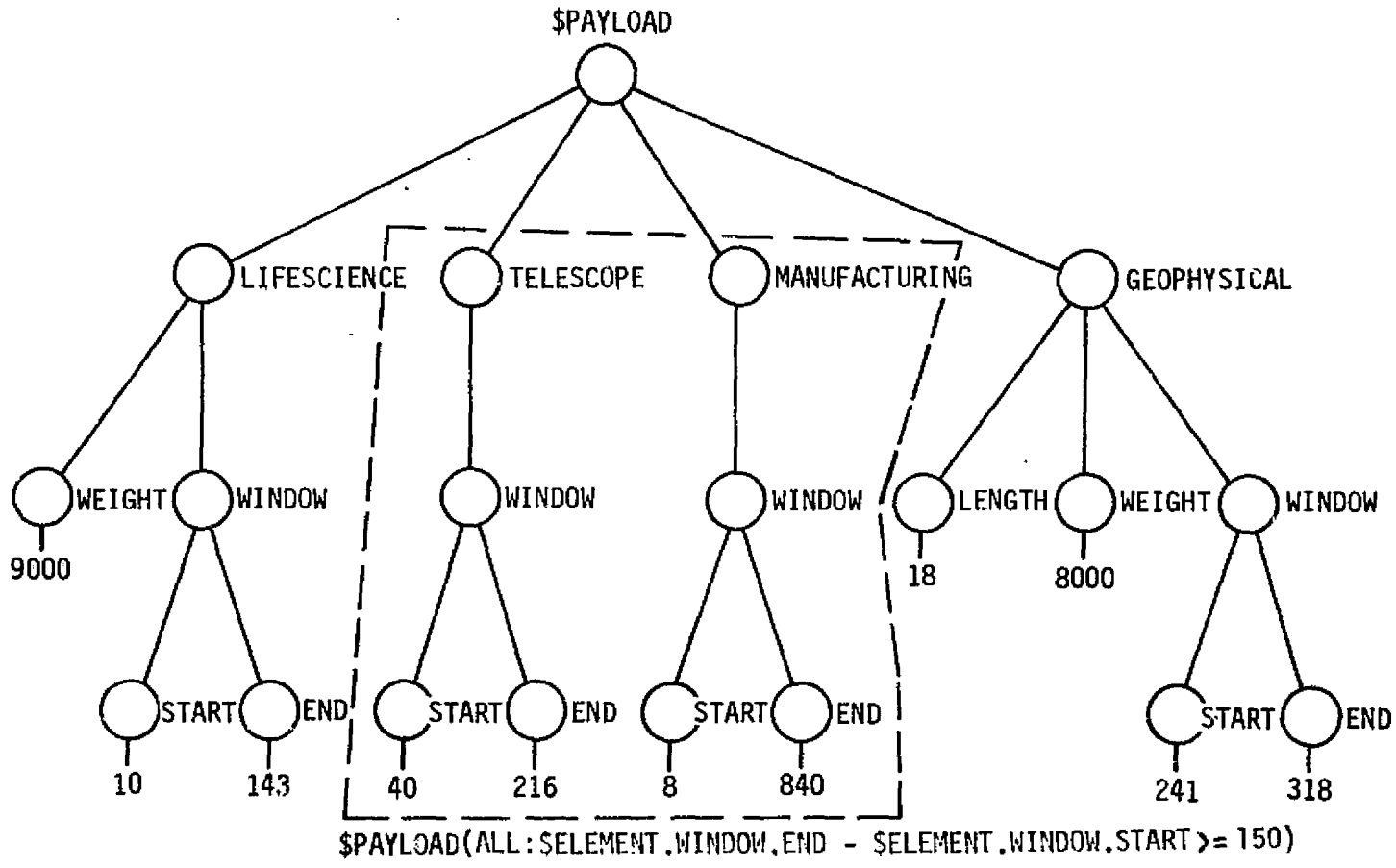


Fig. 2-9 Conditional Access Using Qualifier "ALL:"

An ALL: qualifier must be the last qualifier in a tree reference. Since ALL: constitutes a potential reference to more than one node, it is not possible to qualify it further, so that $PAYLOAD(ALL:$ELEMENT.WINDOW,END - $ELEMENT.WINDOW.START >=150). WEIGHT is an illegal reference.

Since "ALL" is a subscript keyword, "all" cannot appear as a tree node label (i.e. $X.ALL is not allowed).

2.2.8 Indirect Tree Reference

Another powerful access mechanism is the indirect reference, which allows considerable program independence from specific characteristics without loss of efficiency.

For example, unless the programmer resorts to very expensive iterative tree searching there is no way, without indirect referencing, that he can write a program to schedule shuttle flights that does not contain words like PAYLOAD, ORBITER, etc. In order to access information about these resources, he wants to use them as labels for qualified access, or, conceivably, as tree names. What is needed is a capability that allows the characteristics of a problem to reside in the data, rather than the program. Only in this way can a program that schedules shuttle flights also schedule machine shop operations. What the programmer needs is the capability to read in, as data, the labels he will use to access particular tree nodes.

Accordingly, PLANS allows the kind of indirect referencing illustrated in figure 2-10. What the programmer is attempting to do in this illustration is to access information about the resource types named in a tree called $RESOURCE_REQUIREMENTS. He therefore

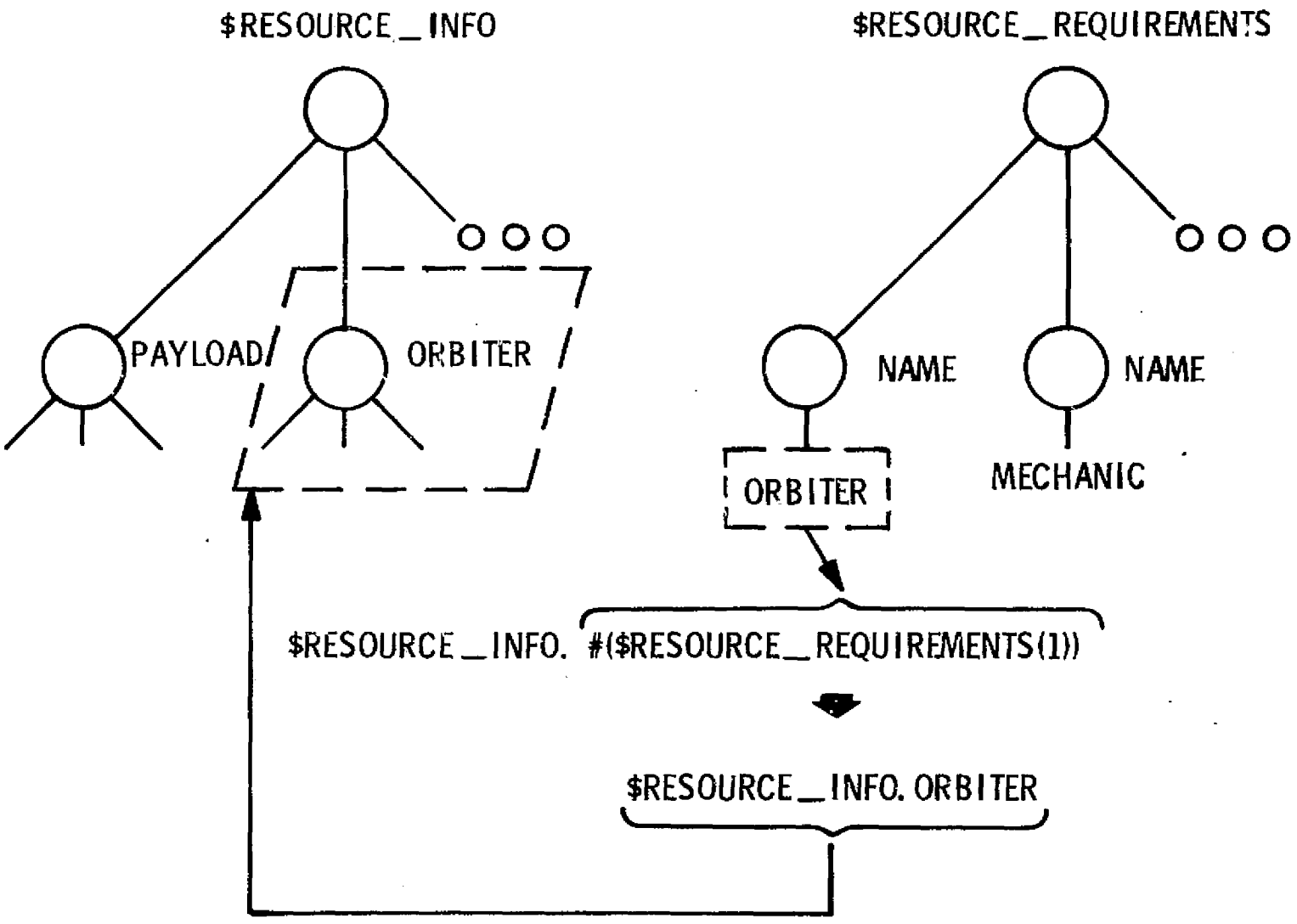


Fig. 2-10 Indirect Referencing

writes the tree node expression, $RESOURCE_INFO.#($RESOURCE_REQUIREMENTS(1)), to access information about the first such resource type. This expression might be read, "the descendant of the node $RESOURCE_INFO whose label is the character string found as the current value of the node $RESOURCE_REQUIREMENTS(1)". The programmer is in effect saying, "Behave as if I had written $RESOURCE_INFO.ORBITER, but allow me the freedom to use some other label than ORBITER by changing the data, without changing the program."

Another type of indirect referencing is illustrated in figure 2-11. Here the programmer is attempting to access information about the resource types whose names appear as labels in the tree $RESOURCE_REQUIREMENTS. He therefore writes the tree node expression, $RESOURCE_INFO.#LABEL($RESOURCE_REQUIREMENTS(1)), to access information about the first such resource type. This expression might be read, "the descendant of the node $RESOURCE_INFO whose label is the character string found as the current label of the node $RESOURCE_REQUIREMENTS(1)".

Indirect referencing can be combined with other qualification methods. The following are valid tree references:

```
$X(2).#($Y.A).START
$X.A.#LABEL($Y.A)(3)
$X.A(FIRST).#($Y(LAST).B).END
$X.#($Y.#LABEL($Z(2)))
```

The last example above shows that indirect referencing can be nested to any depth.

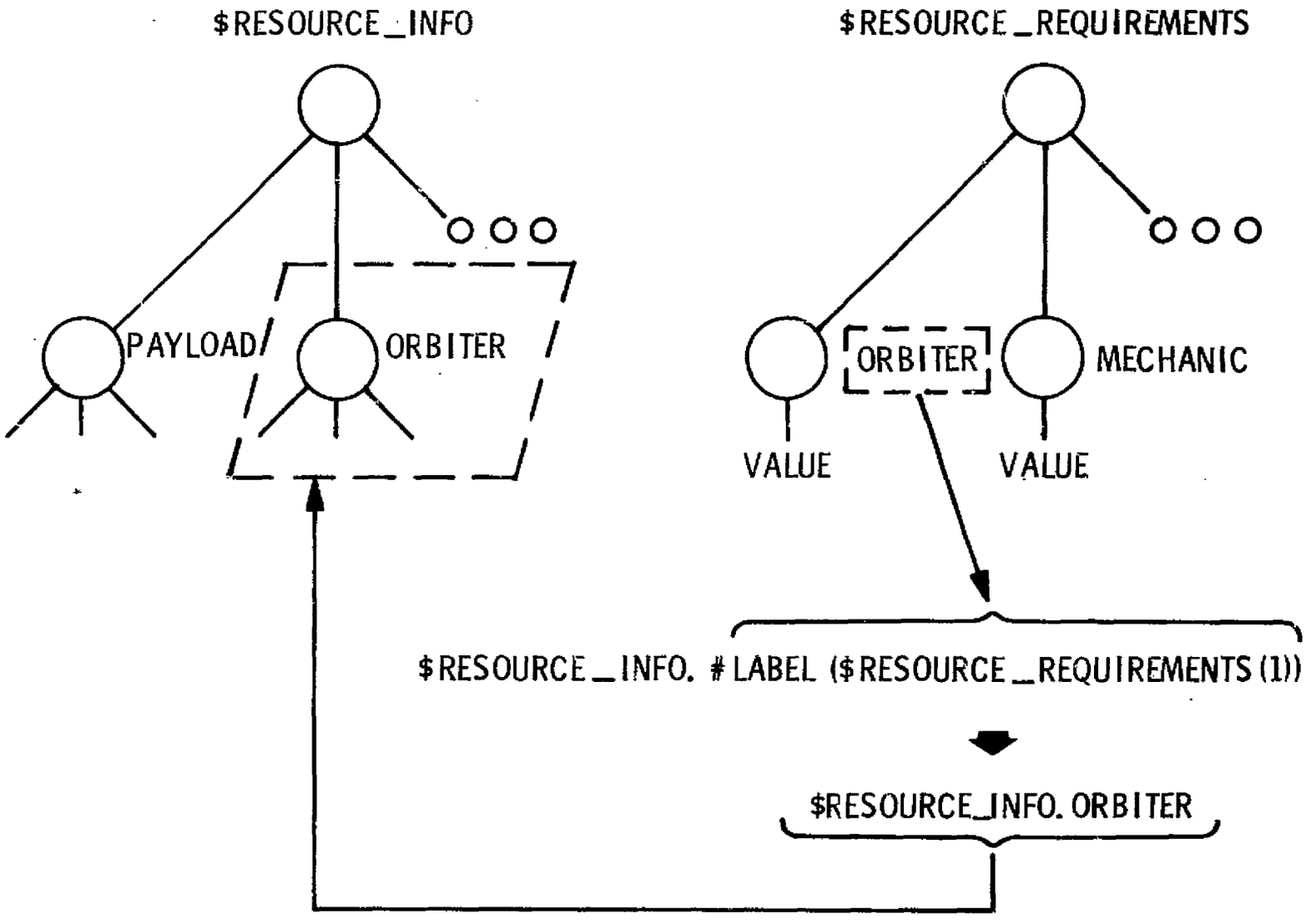


Fig. 2-11 Indirect Referencing (LABEL Option)

2.2.9 Label Function

Since it is sometimes necessary to access the label of a node, the function LABEL was created which takes as its argument a tree node reference. For example, LABEL($X(3)) is a reference not to the node $X(3) and its value or substructure, but to its label alone.

The LABEL function can appear anywhere a character string literal (e.g., 'XYZ') can appear in a PLANS program; LABEL cannot appear as a tree node qualifier (i.e. $X.LABEL and $X.LABEL($Y(1)) are not allowed) unless the tree reference is indirect (i.e., $X.#LABEL($Y(1)) is allowed, see section 2.2.8).

A special application of the LABEL function is in the case of a tree node which has a numeric label. For example, the tree assignment statement (see section 3.1.1) LABEL($X(1)) = 34 causes the first subnode of $X to have a label 34. $X(1) cannot be accessed by label, since $X.34 is not allowed, but we can access $X(1) by $X(FIRST: LABEL($ELEMENT)=34) which utilizes the label 34.

2.2.10 NUMBER Function

A second special function of PLANS is NUMBER. This function returns the number of descendants possessed by a given node. Thus, the expression NUMBER($PAYLOAD) applied to the tree of figure 2-9 yields the numerical value 4.

Since NUMBER is a function keyword, NUMBER cannot appear as the label of a tree node (i.e., $X.NUMBER is not allowed).

2.2.11 Null Nodes and $NULL

If a node has no label and no value or subnodes, it is called a null node. If a node is referred to in a context in which the reference is to the node as a subtree, and the node does not exist, it will be evaluated as a null node. For example, in the tree assignment statement (see section 3.1.1) $X = $Y(ALL:LABEL($ELEMENT)=

'START'), if there are no descendants of $Y which satisfy the condition, $Y(ALL:LABEL($ELEMENT)='START') will refer to a null node, so $X will become null upon execution of this statement.

Often it is desirable for purposes of condition-checking to be able to refer to a node which is assuredly null. The keyword $NULL represents a node which has no label, value, or substructure. If, for example, the programmer wanted to determine whether the tree $X is null, he might write IF $X IDENTICAL TO $NULL THEN... (For an explanation of the boolean relation "IDENTICAL TO", see section 3.4.2.)

2.2.12 Additional Notes Concerning Tree Accesses

The meaning of a tree reference depends to a considerable extent on the context in which it occurs. References to particular nodes may be intended to refer to a tree substructure (i.e., the node "pointed" to, including its label, and anything below that node in the tree) or to a value. The meaning of a tree reference depends on the context in which the reference occurs. Thus, a statement which commands that a node be "pruned" (i.e., deleted from its tree) is obviously a structure reference, while a statement like

```
WINDOW_DURATION = $PAYLOAD(2).WINDOW.END - $PAYLOAD(2).WINDOW.START;
```

refers, because of its arithmetic nature, to the values (216 and 40) of the two tree nodes used in the statement.

If a referenced node does not exist, or has no value, a value is assumed which depends on the context of the reference.

Let us assume, for example, that the node $X has only 2 subnodes. The node $X(3) is therefore a nonexistent node. In an arithmetic context, a reference to this node would be assigned the value zero. For example, after execution of the statement

```
Y = $X(3) + 7;
```

Y will have a value of 7. In a string context, however, the reference evaluates as a null string. Thus, the Boolean expression in

```
IF $X(3) = '' THEN...
```

will be evaluated as true. The same properties are exhibited by a node which exists, but has no value, <u>even though it has a substructure</u>. Thus, in this example, the statement

```
Y = $X + 7;
```

will assign a value of 7 to Y, and the Boolean expression in IF $X = '' THEN ... will be evaluated as true.

2.3 BRIEF LIST OF PLANS STATEMENTS (WITH EXAMPLE STATEMENTS)

Tree Manipulation Statements

Tree Assignment Statement

```
$X(3) = $Y.A;
```

GRAFT Statement

```
GRAFT $NAME(LAST) AT $LIST(NEXT)
```

INSERT Statement

```
INSERT $NEXT_ELEMENT BEFORE $CURRENT_TREE(POSITION);
```

GRAFT INSERT Statement

```
GRAFT INSERT $Z(3) BEFORE $B.X.Y.Z;
```

PRUNE Statement

```
PRUNE $X, $Y(3), $Y(FIRST);
```

Label Assignment Statement

```
LABEL($X(LAST)) = 'SELECTED';
```

ORDER Statement

```
ORDER $PAYLOADS BY WEIGHT;
```

Tree Pointer Statements

DEFINE Statement

```
DEFINE $TREE_POINTER AS $X(CURRENT_INDEX);
```

ADVANCE Statement

```
ADVANCE $TREE_POINTER;
```

Arithmetic Statement

Arithmetic Assignment Statement

```
X = Y*3.0 + 26.5;
```

Conditional Statements and Expressions

IF Statement

```
IF $X IDENTICAL TO $NULL THEN GO TO FINISHED;
```

Boolean Expression

```
X<26.0 | ($Y < MAXIMUM & $Z SUBSET OF $LIST)
```

Control and Transfer of Control Statements

```
GO TO Statement
GO TO TRY_AGAIN;

CALL Statement
CALL ORDER_LIST ($LIST, XMAX, INDEX);

RETURN Statement
RETURN;

STOP Statement
STOP;

TRACE Statement
TRACE HIGH;
```

Input/Output Statements

```
READ Statement
READ $X, $Y, NUMBER_OF_CREWMEN;

WRITE Statement
WRITE  'ERRONEOUS TREE RETURNED',  $RETURNED_TREE
```

Structural Statements

```
PROCEDURE Statement
GET_NEXT_CANDIDATE:  PROCEDURE($CANDIDATE_LIST, $SELECTED);

DECLARE Statement
DECLARE X, $TEMP LOCAL;

BEGIN Statement
BEGIN;

Noniterative DO Statement
DO;

END Statement
END;
```

Iteration Statements

```
DO FOR ALL SUBNODES Statement
DO FOR ALL SUBNODES OF $X USING $POINTER;

DO FOR ALL COMBINATIONS Statement
DO FOR ALL COMBINATIONS OF $CANDIDATES TAKEN 2 AT A TIME;
```

```
DO FOR ALL PERMUTATIONS Statement
DO FOR ALL PERMUTATIONS OF $QUEUE TAKEN NUMBER($QUEUE) AT A TIME;

DO WHILE Statement
DO WHILE ($POINTER NOT IDENTICAL TO $NULL);

Incremental DO Statement
DO I = 1,2,4,7 TO 20;
```

3.0 DETAILED DESCRIPTION OF PLANS STATEMENTS

3.1 TREE MANIPULATION STATEMENTS

There are seven tree manipulation statements in PLANS; each statement allows tree references to an information source and/or an information destination. The seven statements are:

Tree Assignment Statement:

(Destination tree reference) = (Source tree reference);

GRAFT Statement:

GRAFT (Source tree reference) AT (Destination tree reference);

INSERT Statement:

INSERT (Source tree reference) BEFORE (Destination tree reference);

GRAFT INSERT Statement:

GRAFT INSERT (Source tree reference) BEFORE (Destination tree reference);

PRUNE Statement:

PRUNE (Source tree reference), (Source tree reference),..., (Source tree reference);

LABEL Assignment Statement

LABEL (Destination tree reference) = (Expression);

ORDER Statement:

ORDER (Destination tree reference) BY (Property);

The basic tree manipulation statement is the tree assignment statement, which is closely analogous to the ordinary arithmetic assignment statement. In order to provide a context within which to discuss the general properties of tree manipulation statements,

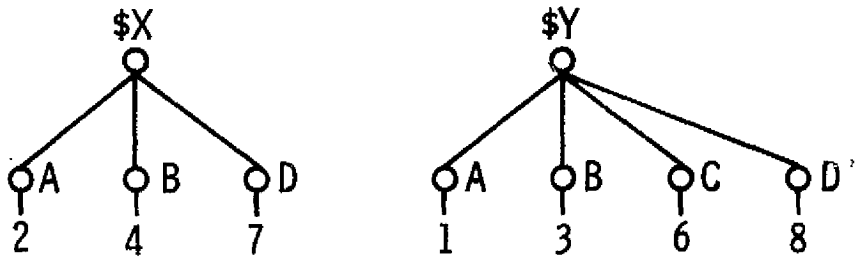
the tree assignment statement will be described in considerable detail, after which the other tree manipulation statements will be discussed in relation to it.

3.1.1 Tree Assignment Statement

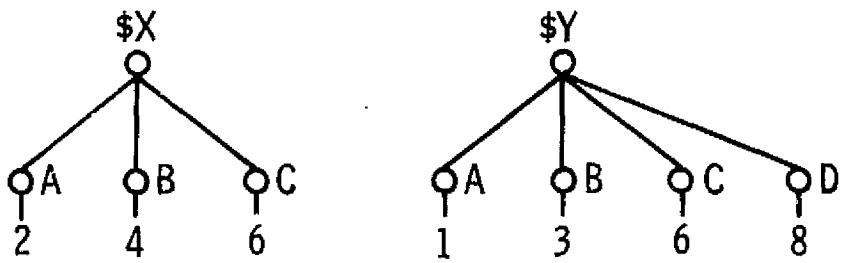
The basic function of the tree assignment statement is to copy tree information (either a node's value or its entire substructure) from one place to another. For example, consider the statement $XTREE = $YTREE;. This statement (1) destroys the current value or substructure of $XTREE, (2) creates a copy of the node and value or substructure of $YTREE, and (3) places the resulting structure at $XTREE. The net result is one of replacement or assignment of a new value.

Of course, the tree node expressions in a tree assignment statement may be more complex than simple tree names. An example is shown in Fig. 3-1, and should be considered in detail. Figure 3-1 (a) shows the initial condition of two trees, $X and SY. The first statement, $X(3) = $Y.C, modifies the tree $X, as shown in (b). Note that the original third subnode of $X has been deleted and replaced with a copy of the node $Y.C, and that the tree $Y has not been altered at all. Note also that the label of $Y.C has replaced the label of $X(3).

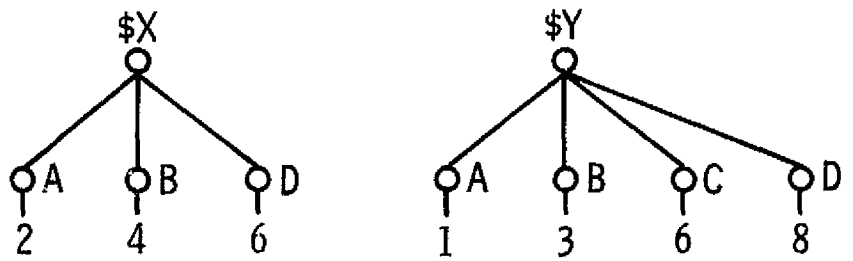
Contrast (b) with (c), where the assignment statement $X.D = $Y.C causes $X.D to be deleted and replaced with a copy of the node $Y.C, which is exactly what occurred with $X(3) = $Y.C, but this time the label of $X(3) remains what it was before the assignment. Thus, in an assignment statement, if the destination already



a) Original Trees



b) Trees After $X(3) = $Y.C



c) Trees of (a) After $X.D = $Y.C

FIG. 3-1 Results of Simple Tree Assignment Statements (Changing Labels and Values)

has a value or substructure, it is automatically deleted before the copy operation occurs. However, the label of the destination will be replaced only if the destination tree reference is qualified by subscript (e.g., $X(3)). If the destination is qualified by label (e.g., $X.D), the label remains the same.

If the destination node does not yet exist, it is created. For example, beginning with the trees in figure 3-2(a), the statement $X(4) = $Y(LAST) results in the modified $X shown in (b). Because the left-hand side of the tree assignment referred to a node not yet in existence, a new subnode of $X was created.

Again beginning with the trees in (a), the statement $X(5) = $Y(LAST) results in the modified $X shown in (c). Note that a null node was created at $X(4) so that a copy of $Y(LAST) could be placed at $X(5).

The destination tree reference is qualified by label in (d), so the tree assignment $X.E = $Y(LAST) results in a modified $X very similar to that in (b) except that now the label of $X(4) is "E". $X.E is created at $X(4) because in tree assignments new subnodes are always added at the right.

Copying a nonexistent or null node results in the generation of a node with no value or substructure at the destination. This case is illustrated in figure 3-3. In (a), there is no node $Y.E. Therefore, the statement $X(2) = $Y.E (1) deletes the contents of $X(2), (2) makes a copy (null) of $Y.E, and (3) replaces $X(2)

a) Original Trees

b) Trees After $X.D = $Y(LAST)

c) Trees of (a) After $X(5) = $Y(LAST)

d) Trees of (a) After $X.E = $Y(LAST)

FIG. 3-2 Results of Simple Tree Assignment Statements (Creating New Nodes)

a) Original Trees

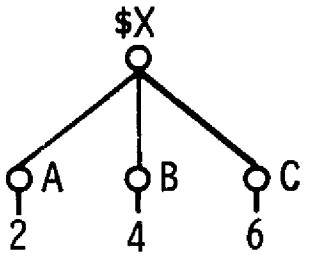
b) Trees After $X(2) = $Y.E

FIG. 3-3 Results of a Tree Assignment Statement When the Source Node is Null

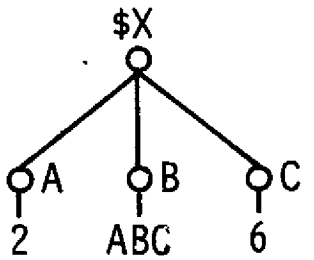
with the copy. That is, $X(2) is replaced by a null node under these circumstances.

This convention is consistent with the execution of the statement when the node in question exists, and has the advantage that it allows the programmer to test explicitly for a null node ("IF $X(2) = $NULL THEN ..."") if he is in doubt about the existence of the node referred to on the righthand side of the tree assignment statement. This same behavior occurs when a conditional tree access is used in which the condition is not satisfied. Suppose, for example, that the programmer had wanted to replace $X(2) in the example by a copy of the first descendant of $Y that had a structure. He might have written $X(2) = $Y(FIRST:NUMBER($ELEMENT) >0). Because none of the descendants of $Y satisfies the condition, the result would have been identical to that resulting from $X(2) = $Y.E. Both statements yield the same result as $X(2) = $NULL.

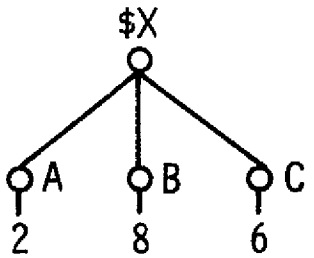
The source information can be arithmetic or a character string rather than a reference to an existing tree node. Type conversion is performed automatically. Figure 3-4(a) shows the initial condition of the tree $X. Figure 3-4(b) shows $X as modified after the execution of the statement $X.B = 'ABC'. Described algorithmically, here is what has happened: (1) the value or substructure of $X.B has been deleted, because $X.B occurs on the left-hand side of a tree assignment statement; (2) the right-hand side of the statement has been evaluated as a tree expression, because that is what is called for by



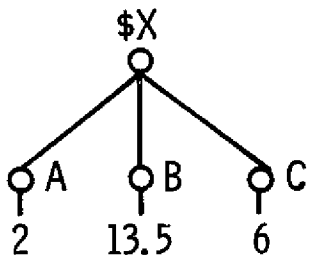
a) Original Tree



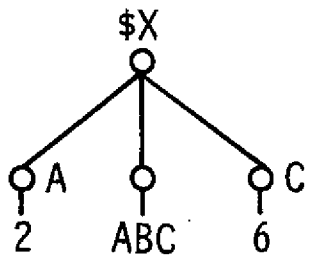
b) Tree After $X.B = 'ABC'



c) Tree of (a) After $X.B = 2*4



d) Tree of (a) After $X.B = $X.C + 7.5



e) Tree of (a) After $X(2) = 'ABC'

FIG. 3-4 Type Conversion in Tree Assignment Statements

the tree assignment statement, and (3) a copy of the tree structure referred to on the right-hand side has replaced the value or substructure of $X.B. By this description, then, this tree assignment statement operated like any other. But how is something evaluated as a tree expression when it is in fact a character string?

Any time a character string or arithmetic expression occurs, when the context clearly calls for a tree expression, a dummy tree is created. This dummy tree has only a single node, the root node, which has a null label. The value of the node is the string or arithmetic value specified in the PLANS expression, in this case the string 'ABC'. The dummy tree is then used just as if the programmer had explicitly created the tree and placed the tree's name in the program. In the case of the example, the result is the same as if the programmer had written $X.B = $DUMMY, where $DUMMY is a tree with one node, no label, and the string value 'ABC'.

As suggested in the explanation above, the same mechanism applies when an arithmetic expression appears in a context that requires a tree node reference. Thus, application of the statement $X.B = 2*4 to the tree of 3-4(a) yields the result shown in (c). The value of the arithmetic expression, in this case 8, is calculated, placed on a dummy node, and becomes the value of $X.B. It may occur to the reader that the same behavior could as well be described as replacement of the value (or substructure) on the left by the value of the expression on the

right. As the discussion of (e) will show, this is not always true. It will be helpful, therefore, to think in terms of the generation of a dummy node when considering statements of this type.

Figure 3-4 (d) shows a statement of the same basic sort as that of (c). In this case, the arithmetic expression on the right-hand side involves a tree node reference. Because $X.C occurs within an arithmetic expression, it has the value 6, just as if $X.C were an arithmetic variable name. Therefore, the statement $X.B = $X.C + 7.5 results in substitution of the numeric value 13.5 at $X.B.

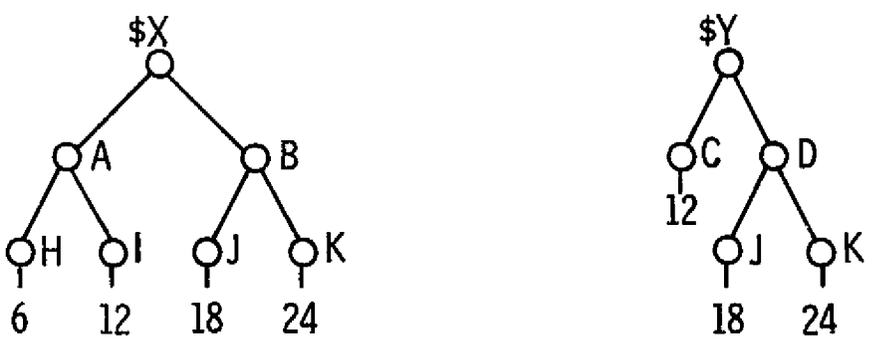
Finally, figure 3-4(e) shows a statement similar to (b), except that the destination tree reference is qualified by subscript. In this case the destination will assume the label of the source tree reference, which is a null label.

A property of PLANS tree operations that should be well understood is the exclusivity of values and substructures. A node may have a null value or it may possess either a value or a substructure, but it may never have both a value and a substructure. Figure 3-5 illustrates this concept. In (b), execution of $X.A = $Y(1) places a new value on the node $X.A, with the result that the previous substructure of $X.A is deleted. Figure 3-5(c) shows the converse case in which placement of a new substructure on the node $X.B deletes the previous value of that node.

Figure 3-6 illustrates some tree assignments to previously non-existent nodes. Figure 3-6 (b) shows the tree of (a) as modified by

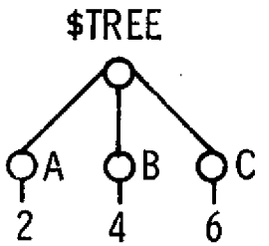
(a) Original Trees

(b) Trees After $X.A = $Y(1)

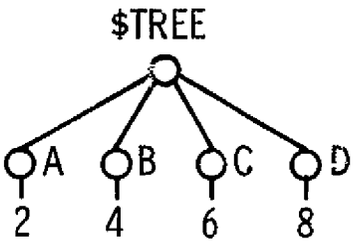


(c) Trees Of (a) After $X.B = $Y(2)

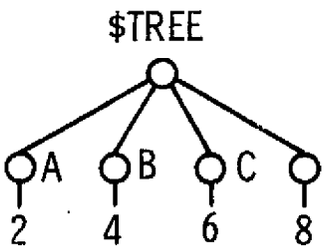
Fig. 3-5 Value-Substructure Exclusivity



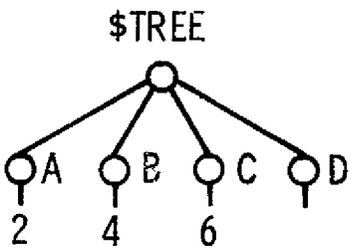
(a) Original Tree



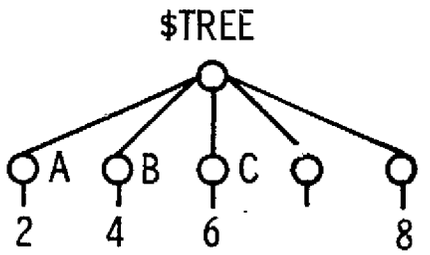
(b) Tree after $X.D = 8



(c) Tree of (a) after $X(NEXT) = 8



(d) Tree of (a) after LABEL ($X(NEXT)) = 'D'



(e) Tree of (a) after $X(5) = 8

Fig. 3-6 Assignments to Nonexistent Nodes

the statement $X.D = 8. It should be noted that this statement has assigned a value and a label to the new node. Any time the referenced node does not exist, it is caused to exist as specified. If it was specified by label, this means the indicated label must be placed on the new node.

Figure 3-6(c) shows the result of a statement in which a non-existent node was specified by subscript. Because no label was used to indicate the node and the expression on the right has no label (it is a dummy node), the resulting node has a value, but no label. Figure 3-6 (d) involves a new node with a label, but no value. In the figure this result was achieved by the statement LABEL($X(NEXT)) = 'D'. However, because two prime (quote) marks together refer to the null character string, the same result would be observed after execution of the statement $X.D = ''. This statement places a null string on the node as a value, but that is completely equivalent to no value at all.

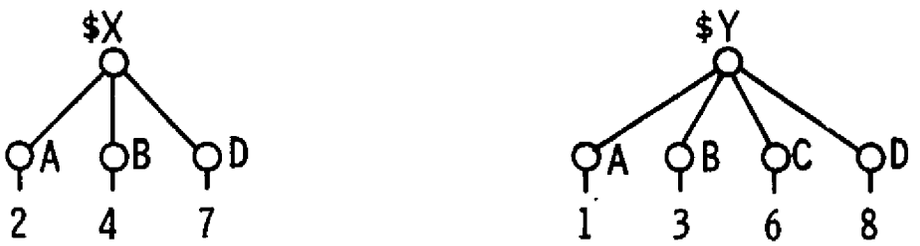
Figure 3-6(e) shows what happens when an assignment is made to a node specified by a subscript that is too large. (It is, of course, only too large if the programmer did not want the result shown in the figure.) The programmer has stated that the fifth subnode of $X is to acquire the value 8. But this can only occur if, after execution of the statement, $X has at least five descendants. Because there were only three descendants before the statement was executed, two new nodes will be created. Only the latest of these newly created nodes is involved in a tree assignment statement; therefore, only the last node can acquire a label or a value. The remaining new node(s), in this case $X(4), will be null.

There need be no correspondence of level between the source and destination references. Since the tree assignment statement just copies the substructure or value at the source and places this copy at the destination, no reference is made to tree levels at all. Notice also that the source node is not affected by the execution of a tree assignment statement.

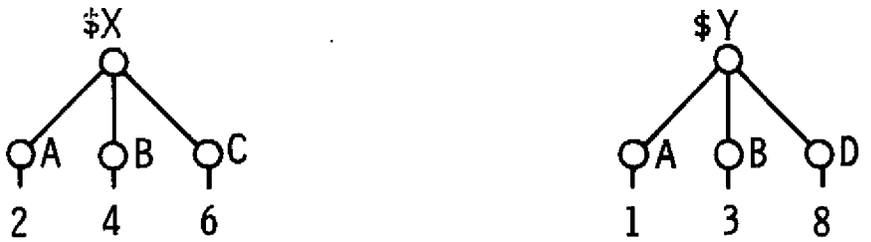
3.1.2 GRAFT Statement

The GRAFT statement, as its name suggests, involves the removal of a piece of one tree which is then placed on another tree. Note that instead of copying the information to be added to the target tree, as is done in tree assignment statements, the GRAFT statement removes the specified structure from its original location and moves it to the target tree. Examples are shown in figure 3-7. Figures (a)-(d) illustrate the fact that the tree assignment and GRAFT statements have the same effect (replacement) on $X, the destination reference. However $Y, the source reference, remains unaffected by a tree assignment statement while it is altered by a GRAFT statement.

It is sometimes important to recognize that the execution of a GRAFT statement is a sequential process, involving first the removal of a tree substructure from the source tree, then the placement of the structure at the destination. Figure 3-7(e) illustrates the effects of GRAFT $X.C at $X(4). First, $X.C is removed from the source tree, (note that $X has only three subnodes now); then $X.C is placed at $X(4). But since $X has only three subnodes after the removal of $X.C, $X.C is placed to the right of $X.D.



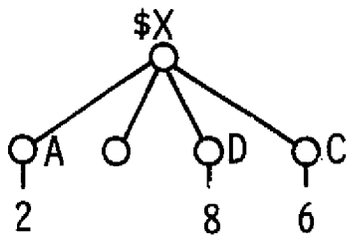
(a) Original Trees



(b) Trees After GRAFT $Y.C AT $X(3)

(c) Trees Of (b) After GRAFT $Y(LAST) AT $X.D

(d) Trees Of (c) After GRAFT $Y.E AT $X(2)



(e) Tree Of (d) After GRAFT $X.C AT $X(4)

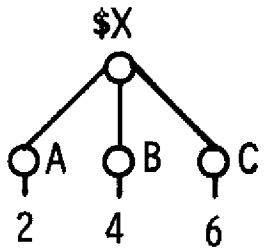
Fig. 3-7 GRAFT Statements

Because it does not involve any copying of tree nodes, the GRAFT statement executes much more quickly than the tree assignment statement. GRAFT gives the appearance of being more complex than the tree assignment statement, and the programmer may naturally assume that the latter is more efficient and should be given preference whenever there is a choice. A little reflection on the underlying structural operations will show that this is not true.

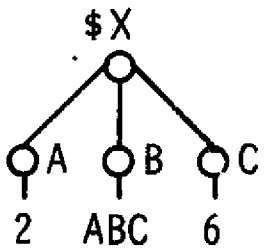
The tree assignment statement requires the generation of a complete copy of an existing structure. The execution cost of these statements (and the storage space required) is largely a function of the size of the structure that must be copied. The GRAFT statement, on the other hand, requires only the alteration of a few pointers so that an existing structure can be moved, completely intact, to another tree location. The execution cost of this statement is minimal, no additional storage is involved, and the cost is entirely independent of the size of the structure that is moved. It cannot be overemphasized that the GRAFT operation is not only very powerful, but also very efficient.

The source information may be of an arithmetic or character string type, in which case the statement behaves like the corresponding tree assignment statement. Figure 3-8 illustrates GRAFT statements which are equivalent to the tree assignment statements in figure 3-4.

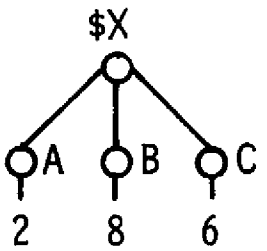
Similarly, if the source reference is to a nonexistent node, the GRAFT statement has the same effect as the corresponding tree assignment statement. Figure 3-7(d) illustrates the effect of the statement GRAFT



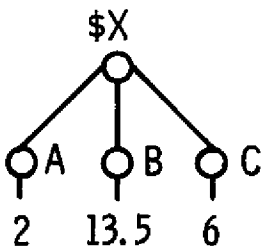
(a) Original Tree



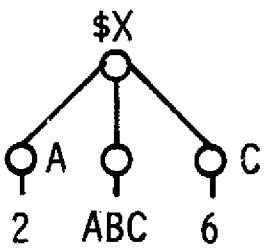
(b) Tree After GRAFT 'ABC' AT $X.B



(c) Tree Of (a) After GRAFT 2*4 AT $X.B



(d) Tree Of (a) After GRAFT $X.C +7.5 AT $X.B



(e) Tree Of (a) After GRAFT 'ABC' AT $X(2)

Fig. 3-8 GRAFT Statements With String And Arithmetic Source References.

$Y.E AT $X(2). Since $Y.E is nonexistent, a null node is created and placed at $X(2).

Like the tree assignment statement, the GRAFT statement may or may not result in replacement of the label of the destination node. If the destination reference is by label, no label replacement will occur; if by subscript, the label will be replaced.

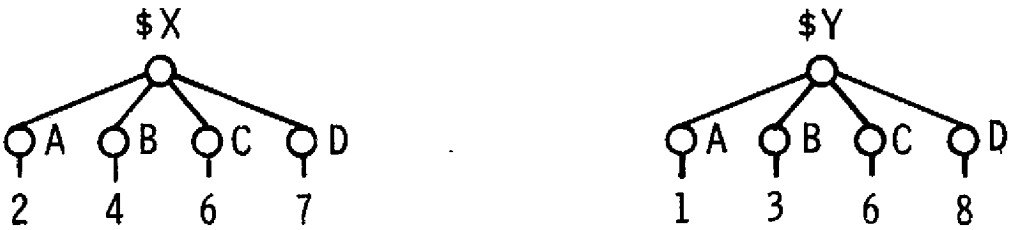
3.1.3 INSERT Statement

For the purpose of the INSERT statement, the subnodes of a node are regarded as an ordered list. Rather than replacing one of the elements of that list, this statement inserts a copy of the source tree before one of them. Examples are shown in Figure 3-9. Like the tree assignment statement, a copy is made of the source reference, but an INSERT statement places this copy before the destination node.

It should be observed that the INSERT operation of (d) results in two subnodes of $X that possess the same label. This is quite allowable, but the programmer should be aware that, if this occurs, the subsequent reference $X.D is a reference to only the first such node. Either node can still be referenced by subscript, however, and a reference of the form $X(ALL: LABEL($ELEMENT) = 'D') would access all such nodes in one operation.

Type conversion occurs automatically if the source reference is arithmetic or a character string. In an INSERT statement, as with the arithmetic assignment statement, any time a character string or

a) Original Trees



b) Trees after INSERT $Y.C BEFORE $X (3)

c) Trees of (a) after INSERT $Y.C BEFORE $X.D

d) Trees of (c) after INSERT $Y(LAST) BEFORE $X.D

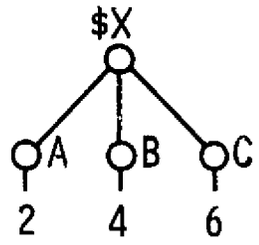
Fig. 3-9 Results of a Sequence of INSERT Statements

arithmetic expression occurs, when the context clearly calls for a tree expression, a dummy tree is created. This dummy tree has only a single node, the root node, which has a null label. The value of the node is the string or arithmetic value specified in the PLANS expression, for example the string 'ABC' in the figure 3-10(b). The dummy tree is then used just as if the programmer had explicitly created the tree and placed the tree's name in the program. In the case of the example, the result is the same as if the programmer had written INSERT $DUMMY BEFORE $X.B, where $DUMMY is a tree with one node, no label, and the string value 'ABC'.

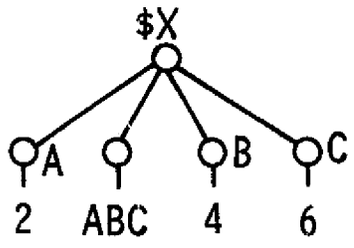
As suggested in the explanation above, the same mechanism applies when an arithmetic expression appears in a context that requires a tree node reference. Thus, application of the statement INSERT 2*4 BEFORE $X.B to the tree of 3-10(a) yields the result shown in (c). The value of the arithmetic expression, in this case 8, is calculated, placed on a dummy node, and becomes the value of the node to the left of $X.B.

Figure 3-10(d) shows a statement of the same basic sort as that of (c). In this case, the arithmetic expression on the right-hand side involves a tree node reference. Because $X.C occurs within an arithmetic expression, it has the value 6, just as if $X.C were an arithmetic variable name. Therefore, the statement INSERT $X.C+7.5 BEFORE $X.B results in insertion of the numeric value 13.5 before $X.B.

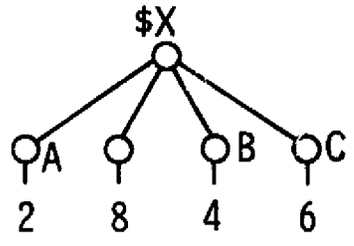
Neither the source node nor the destination node is directly affected by the INSERT operation. In particular, the question of replacement or nonreplacement of the destination base node label does not arise with insertion. Note that the statements, "INSERT $Y.C BEFORE $X.D" and "INSERT $Y.C BEFORE $X(3)" are identical (see (b) and (c) in figure 3-9).



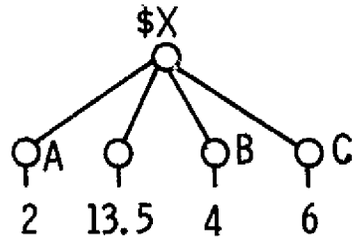
(a) Original Tree



(b) Tree After INSERT 'ABC' BEFORE $X.B



(c) Tree Of (a) After INSERT 2*4 BEFORE $X.B



(d) Tree Of (a) After INSERT $X.C + 7.5 BEFORE $X.B

Fig. 3-10 Type Conversion In INSERT Statements

If the source reference is to a null or nonexistent node, a null node is created and inserted at the indicated location. Figure 3-11(b) illustrates this point.

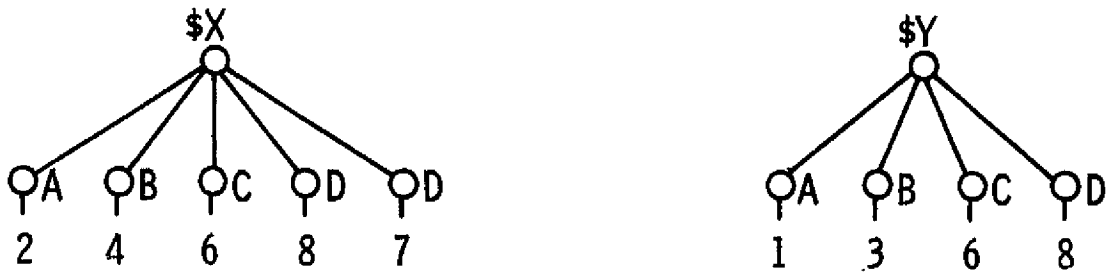
If the destination node exists but is null, funny things happen. It is assumed that the PLANS programmer does not ordinarily leave null nodes lying about. For example, figure 3-11(c) illustrates the statement INSERT $Y.A BEFORE $X(2), where $X(2) is a null node. Note that the copy of $Y.A is not placed before $X(2), but right on the null node.

If the destination node does not exist, the INSERT statement behaves like the tree assignment statement. Figure 3-12 illustrates the statements, $X(4)=$Y.B and INSERT $Y.B BEFORE $X(4). Note that the trees resulting from the two statements are identical.

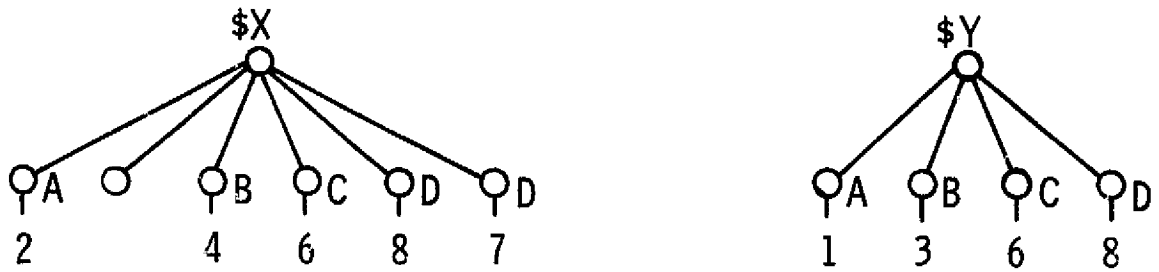
3.1.4 GRAFT INSERT Statement

The GRAFT INSERT statement is a simple combination of the properties of the INSERT statement with those of the GRAFT statement. It possesses all the basic properties of INSERT and, in addition, the source node is removed from its tree. Examples are illustrated in figure 3-13.

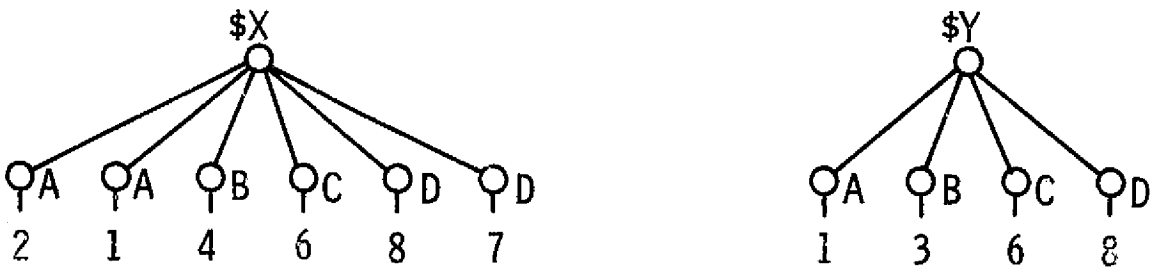
As with the INSERT operation, it should be observed that the GRAFT INSERT operation of (c) results in two subnodes of $X that possess the same label. This is quite allowable, but the programmer should be aware that, if this occurs, the subsequent reference $X.D is a reference to only the first such node. Either node can still be referenced by subscript, however, and a reference of the form



(a) Original Trees

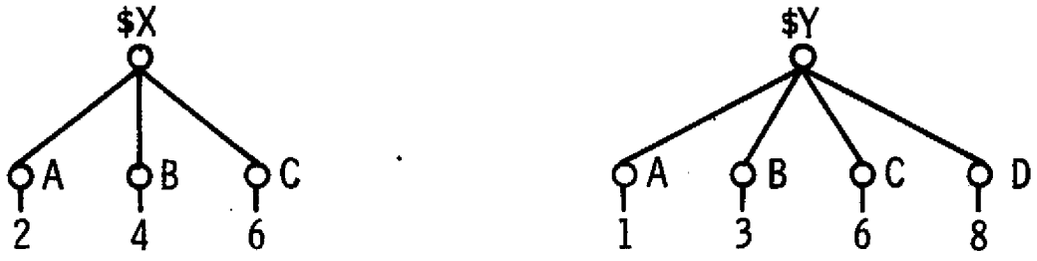


(b) Trees After INSERT $Y.E BEFORE $X(2)

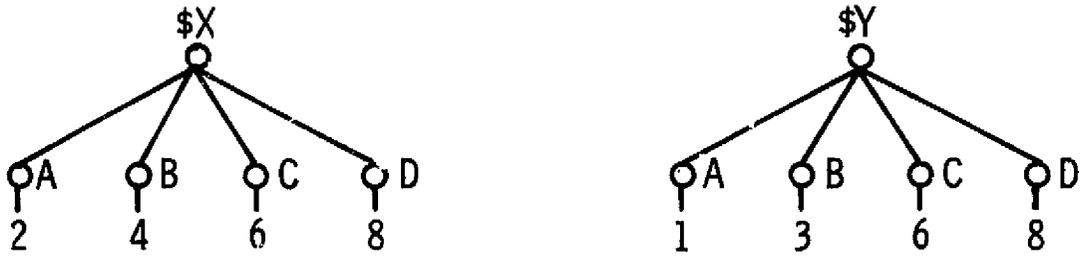


(c) Trees Of (b) After INSERT $Y.A BEFORE $X(2)

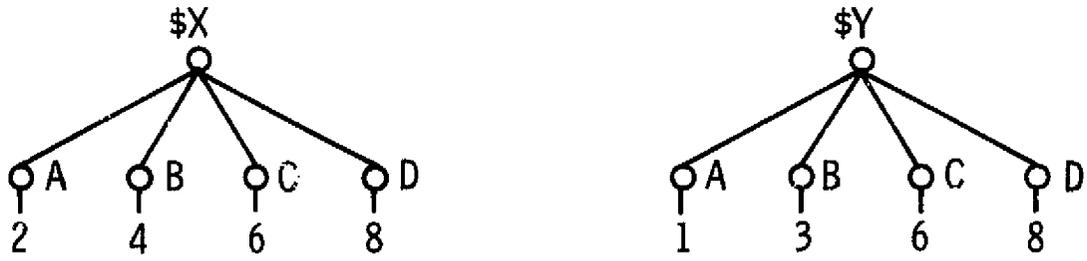
Fig. 3-11 INSERT Statements With NULL Source And Destination Nodes



a) Original Trees

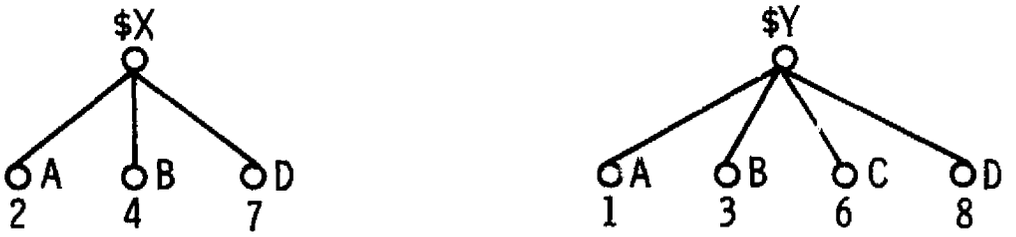


b) Trees of (a) After $X(4) = $Y.D

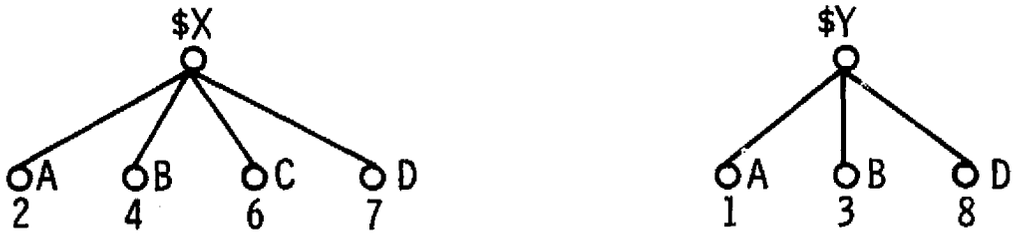


c) Trees of (a) After INSERT $Y.D BEFORE $X(4)

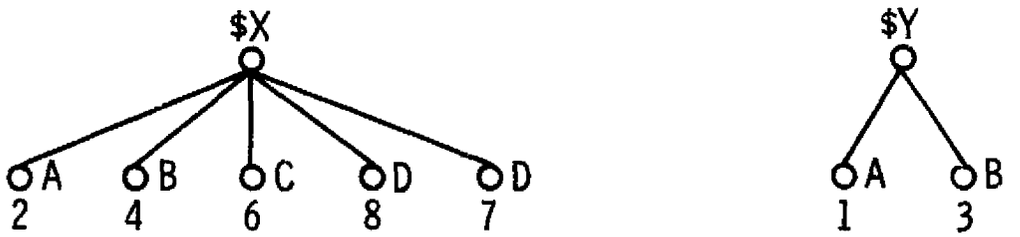
FIG 3-12 INSERT Statement With A Non-Existent Destination Node



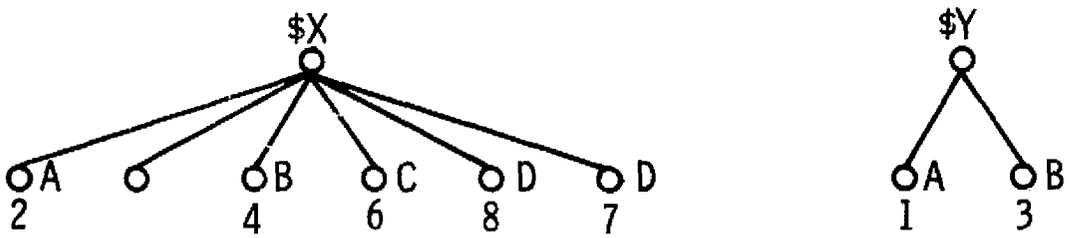
a) Original Trees



b) Trees After GRAFT INSERT $Y.C BEFORE $X(3)



c) Trees of (b) After GRAFT INSERT $Y (LAST) BEFORE $X.D



d) Trees of (c) After GRAFT INSERT $Y.E BEFORE $X(2)

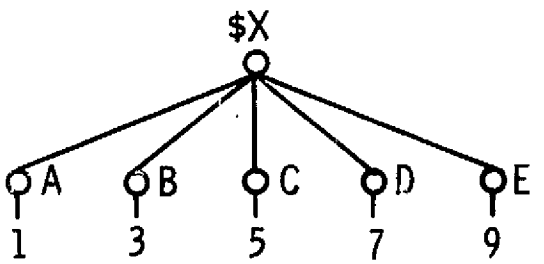
FIG 3-13 GRAFT INSERT Statements

$X(ALL:LABEL($ELEMENT) = 'D') would access all such nodes in one operation.

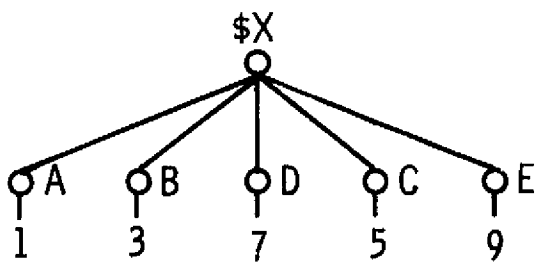
Like the GRAFT statement, the GRAFT INSERT statement is executed in a sequential manner which should be understood. The GRAFT INSERT statement first removes the specified structure from its original location, then inserts it before the destination location.

Thus, the statement, GRAFT INSERT $X.C BEFORE $X(4) results in the tree of figure 3-14(b). Note that first $X.C is removed from the tree; then $X.C is placed before $X(4), but $X(4) is now the node $X.E (i.e., after the removal of $X.C, $X(1) is $X.A, $X(2) is $X.B, and X(3) is $X.D, and $X(4) is $X.E).

As with an INSERT statement, in a GRAFT INSERT statement any time a character string or arithmetic expression occurs when the context clearly calls for a tree expression, a dummy tree is created. This dummy tree has only a single node, the root node, which has a null label. The value of the node is the string or arithmetic value specified in the PLANS expression, for example the string 'ABC' in figure 3-15(b). The dummy tree is then used just as if the programmer had explicitly created the tree and placed the tree's name in the program. In the case of the example, the result is the same as if the programmer had written GRAFT INSERT $DUMMY BEFORE $X.B, where $DUMMY is a tree with one node, no label, and the string value 'ABC'.

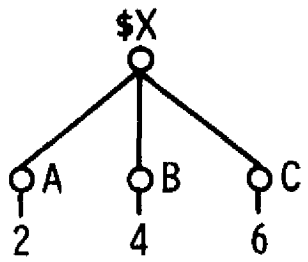


a) Original Tree

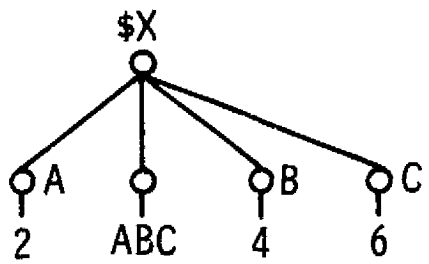


b) Tree After GRAFT INSERT $X.C BEFORE $X(4)

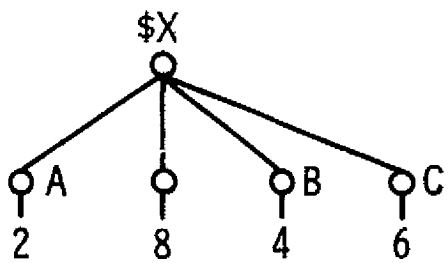
FIG 3-14 Sequential Execution of a GRAFT INSERT Statement



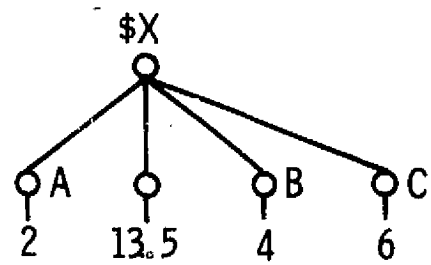
a) Original Tree



b) Tree After GRAFT INSERT 'ABC' BEFORE $X.B



c) Tree of (a) After GRAFT INSERT 2*4 BEFORE $X.B



d) Tree of (a) After GRAFT INSERT $X.C + 7.5. BEFORE $X.B

FIG 3-15 Type Conversion in GRAFT INSERT Statements

As suggested in the explanation above, the same mechanism applies when an arithmetic expression appears in a context that requires a tree node reference. Thus, application of the statement GRAFT INSERT 2*4 BEFORE $X.B to the tree of figure 3-15(a) yields the result shown in (c). The value of the arithmetic expression, in this case 8, is calculated, placed on a dummy node, and becomes the value of the node to the left of $X.B.

Figure 3-15(d) shows a statement of the same basic sort as that of (c). In this case, the arithmetic expression on the right-hand side involves a tree node reference. Because $X.C occurs within an arithmetic expression, it has the value 6, just as if $X.C were an arithmetic variable name. Therefore, the statement GRAFT INSERT $X.C +7.5 BEFORE $X.B results in insertion of the numeric value 13.5 before $X.B.

GRAFT INSERT, like GRAFT, involves no copying and is relatively efficient to execute. The tree assignment and INSERT statements require the generation of a complete copy of an existing structure. The execution cost of these statements (and the storage space required) is largely a function of the size of the structure that must be copied. The GRAFT and GRAFT INSERT statements, on the other hand, require only the alteration of a few pointers so that an existing structure can be moved, completely intact, to another tree location. The execution cost of these statements is minimal, no additional storage is involved, and the cost is entirely independent of the size of the

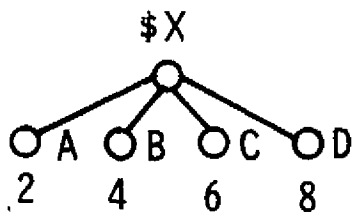
structure that is moved. It cannot be over-emphasized that GRAFT and GRAFT INSERT operations are not only very powerful, but are also very efficient.

3.1.5 PRUNE Statement

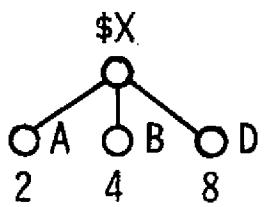
The function of the PRUNE statement is to remove one or more specified nodes (with substructures, if any) from their trees. Examples are illustrated in figure 3-16. The programmer simply specifies the node (or nodes) that, together with the associated substructure, is to be removed. This operation allows the removal of undesired information from a tree. It may also be used, particularly as in (d), to release computer storage that is no longer needed.

It should be kept in mind while programming in PLANS that the programmer is really doing his own dynamic storage allocation (although PLANS handles all the details for him). When information is no longer needed, its storage can be reused, but only if the programmer releases it by means of a PRUNE statement.

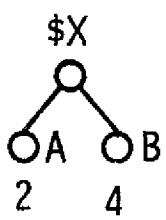
If a PRUNE statement refers to more than one node to be pruned, it must be kept in mind that the pruning operations occur in sequence, one after another. An example of this is the seemingly redundant statement, PRUNE $X(2), $X(2), illustrated in figure 3-17. Initially, $X(1) is $X.A, $X(2) is $X.B, $X(3) is $X.C, and $X(4) is $X.D; so $X.B is pruned. Now $X(1) is $X.A, $X(2) is $X.C, and $X(3) is $X.D; so $X.C is also pruned.



(a) Original Tree



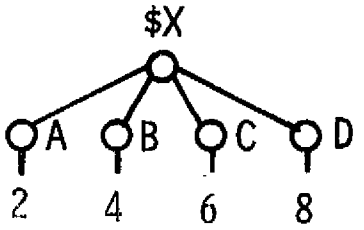
(b) Tree after PRUNE $X.C



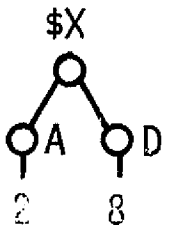
(c) Tree of (a) after PRUNE $X.C, $X.D

(d) Tree of (a) after PRUNE $X

Fig. 3-16 Prune Statements



a) Original Tree



b) Tree after PRUNE $X(2), $X(2)

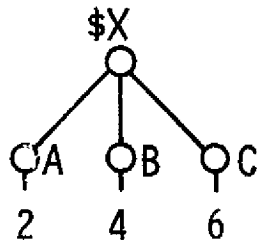
Fig. 3-17 Sequential Execution of the PRUNE Statement

3.1.6 Label Assignment Statement

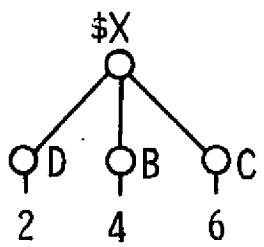
The label assignment statement replaces only the label of the indicated node, without disturbing its value or substructure. LABEL is a special PLANS function that takes as its argument a tree node reference. LABEL ($X(1)) is a reference not to the node $X(1) and its substructure, but to its label alone. The LABEL function can appear anywhere a character string can appear in a PLANS program. In addition, it can appear on the left-hand side of an equal sign, as figure 3-18 shows. Such a statement is a command to replace the current label of the specified node with the new string, which is obtained by evaluating the expression to the right of the equal sign.

Consider figure 3-18; (a) shows the initial state of the tree $X. Figure 3-18(b) illustrates the effect of the label assignment statement LABEL($X(1)) = 'D'; which simply replaces the current label of the node $X(1), "A", with a new string, "D". The label assignment statement only replaces labels, having no structural effect if the referenced node already exists. If the indicated node does not exist, it is established, with a null value and the indicated label.

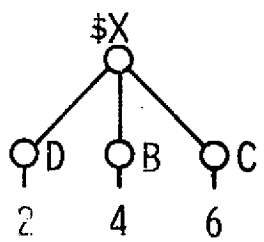
The statement illustrated in (c) has exactly the same effect as that of (b). It makes no difference whether the node is referenced by label ($X.A) or by subscript ($X(1)). The effect of the statement is the same.



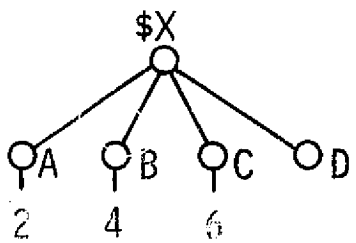
(a) Original Tree



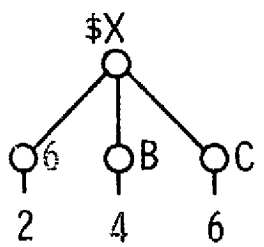
(b) Tree After LABEL ($X(1)) = 'D'



(c) Tree Of (a) After LABEL ($X.A) = 'D'



(d) Tree Of (a) After LABEL ($X.E) = 'D'



(e) Tree Of (a) After LABEL ($X(1)) = $X.C

Fig. 3-18 LABEL Assignment Statements

If the referenced destination node does not yet exist, it is established. Figure 3-18 (d) illustrates the statement, LABEL ($X.E) = 'D', where $X.E is a nonexistent node.

If the source information is not a character string, automatic type conversion occurs. Figure 3-18(e) is an illustration of automatic conversion. The right-hand side of the statement LABEL ($X(1)) = $X.C is a tree expression, but the context calls for a string or numerical value. The value of $X.C is therefore obtained, and replaces the label of $X(1).

An additional concept is illustrated here: labels can be numerical values. In fact, anything that can be a value can be a label, and vice versa. However, nodes that have numerical values (or strings not having identifier syntax) cannot be accessed by label in a PLANS program. Thus, $X.6 is not a legal expression. On the other hand, $X(1) is still a legitimate way to refer to this node. This property can be used to advantage in some numerical applications.

3.1.7 ORDER Statement

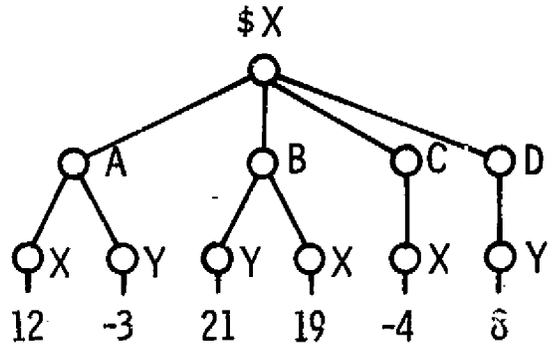
The ORDER statement is used to reorder the subnodes of a given node on the basis of a numerical property that each possesses. If, for example, it is desired to order a group of payloads by weight, heaviest first, one might write ORDER $PAYLOADS BY WEIGHT; or if they were to be ordered by length, longest first, a statement of the form ORDER $PAYLOADS BY LENGTH would be appropriate.

Figure 3-19 illustrates a few of the properties of the ORDER statement. It should be apparent that an ORDER statement refers to a node, which, in turn, has subnodes to be ordered. Each subnode has, at least potentially, the property or properties on which ordering is to occur.

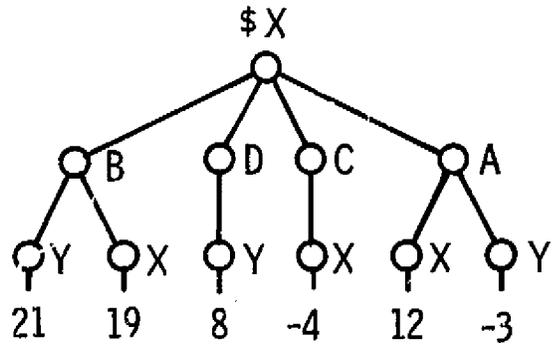
Figure 3-19(b) illustrates an ORDER statement in which the subnodes of the node $X are sorted into descending order on the basis of property Y. Note that, where the property in question is not possessed by a particular subnode (e.g., $X.C has no subnode labeled Y), a value of zero is assumed and the sort is performed accordingly. Note also that the normal ordering is descending; that is, the largest value occurs first. Thus, after execution of ORDER $PAYLOADS BY WEIGHT, the heaviest payload will be $PAYLOAD(1).

More than one property can be listed in order to cause ties on the first property to be broken by values on the second, etc. For example, if it is desired to order a group of payloads by length, longest first, with ties broken by width, widest first, a statement of the form ORDER $PAYLOADS BY LENGTH, WIDTH would be appropriate.

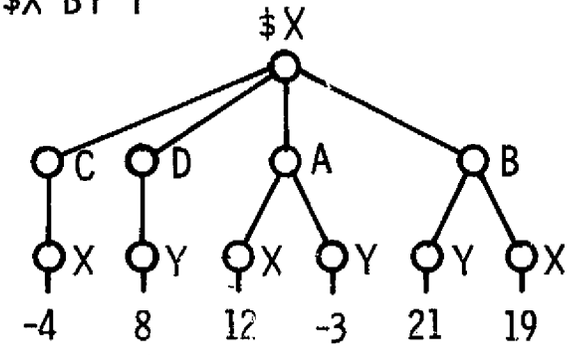
Ordering can be in either ascending or descending sequence. Figure 3-19(b) illustrates ordering in descending sequence (i.e., largest value of Y first). If ascending order is desired, the property name should be preceded by a minus sign (-), as in (c). Thus, a statement like ORDER $PAYLOADS BY -LENGTH, -WIDTH would order payloads by length, shortest first, with ties broken by width, narrowest first. An example is illustrated in figure 3-19(c).



(a) Original Tree



(b) Tree after ORDER $X BY Y



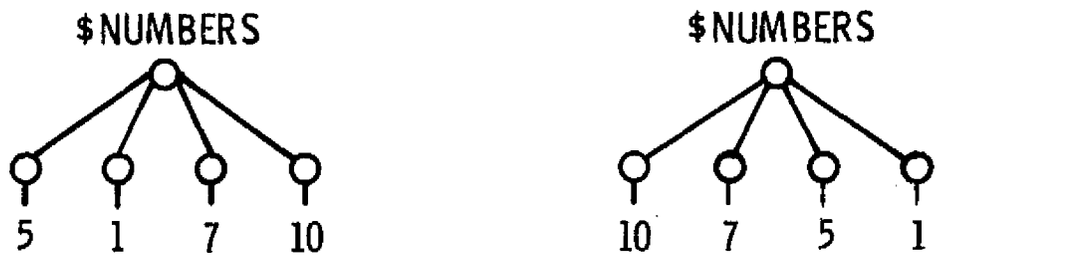
(c) Tree after ORDER $X BY -X

Fig. 3-19 ORDER Statements

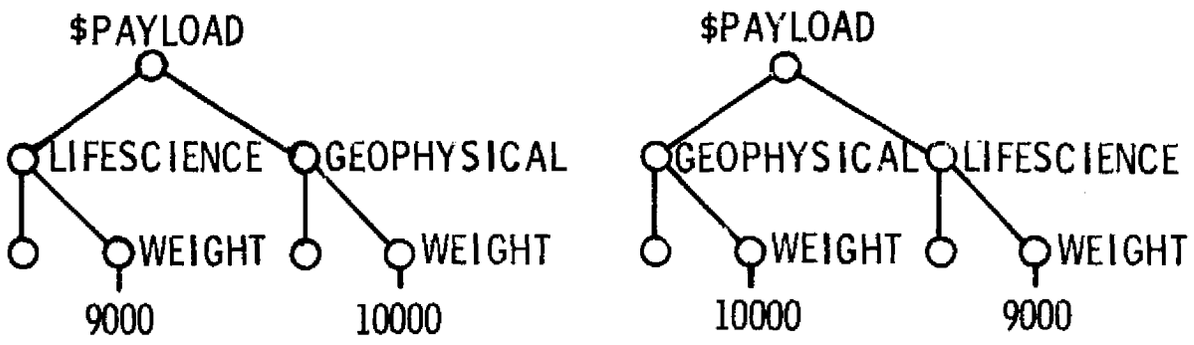
The tree pointer (see section 3.2) $ELEMENT may be used in cases where ORDER (tree node reference) BY (common label of ordering nodes) cannot be applied; for example, a tree with no labels or with no common labels on the ordering nodes can be ordered by the statement, ORDER (tree node reference) BY $ELEMENT (see figure 3-20(a)). Note that $ELEMENT always refers to the nodes being ordered. Thus, the statement illustrated in (a) orders the subnodes of $NUMBERS by their values.

$ELEMENT may be qualified by subscript or label. Thus, to order the subnodes of $PAYLOAD by their weights, we might use the statement, ORDER $PAYLOAD BY $ELEMENT.WEIGHT; or, equivalently, ORDER $PAYLOAD BY $ELEMENT(2) (see figure 3-20(b)). Note that the former statement is equivalent to ORDER $PAYLOAD BY WEIGHT.

Figure 3-20(c) illustrates the basic difference between ORDER (tree node reference) BY $ELEMENT and ORDER (tree node reference) BY (common label of ordering nodes). Note that figure (c) can be ordered by the statement ORDER $X BY $ELEMENT, but it cannot be ordered by the statement ORDER $X BY Y. The latter statement implies that each of the subnodes of $X (these subnodes are the nodes to be ordered) has in turn a subnode labelled Y. If such a statement is applied to the tree of figure (c), each of the values on which ordering is to occur will be zero, and the intended ordering will not occur.

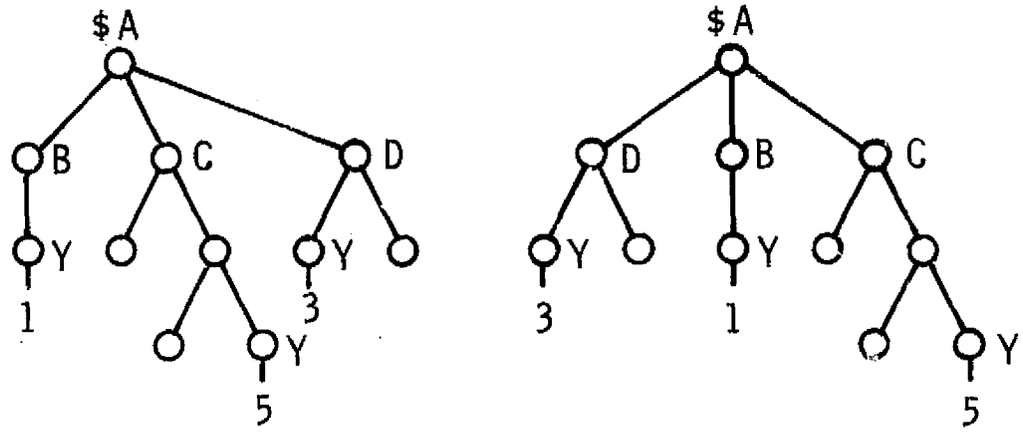


a) ORDER $NUMBERS BY $ELEMENT



b) ORDER $PAYLOAD BY $ELEMENT.WEIGHT

c) ORDER $X BY $ELEMENT (Note: ORDER $X BY Y NOT VALID)



d) ORDER $A BY Y

Fig. 3-20 ORDER Statement Using the Tree Pointer $ELEMENT

Ordering can be on the basis of the values of the sorted nodes, or on properties any number of levels down. Figure 3-20(c) is an example of the first type of ordering, and figure 3-20(b) is an example of the second type.

Note that figure (d) illustrates a tree which cannot be ordered with respect to Y since Y is on a different level for each node to be ordered. Since the first Y-labelled node encountered is on the third-level, the other Y-labelled nodes will be expected to be on the third level. If the statement, ORDER $A BY Y is executed on this tree, $A.C will be assumed to have a Y-value of zero.

3.2 TREE POINTER STATEMENTS

3.2.1 DEFINE Statement

The standard form of this statement is:

DEFINE tree-pointer-name AS tree-node-reference;

The DEFINE statement is used to set a tree pointer so that it can be used to refer to a specified tree node. This capability allows the PLANS programmer to reference any tree node simply by using the tree pointer name. For instance, if the user wished to refer to the fifth subnode of a tree called $SAMPLE_TREE he might write DEFINE $SUBNODE_5 AS $SAMPLE_TREE(5); Thereafter, $SUBNODE_5 could be used to refer to whatever node happened to be $SAMPLE_TREE(5) at the time the DEFINE statement was executed.

It is important to note that the DEFINE statement causes an action to occur at execution time, and not at compile time. That is, the DEFINE statement can be used to dynamically update tree pointers and is not intended to be used for defining static storage "equivalences." The data attributes of a tree pointer are different from those of a "real" tree. Due to this fact, an attempt to use the same tree name for both purposes will cause an error.

Logically, tree pointers can only indicate already-existent tree nodes. Therefore, if the node referred to in the DEFINE statement does not already exist, it will be generated when the statement is executed.

Considerable improvement in both program readability and execution efficiency can be realized through the use of tree pointer statements. Using a long list of qualifiers (labels, subscripts, etc.) to access a subnode several levels down in a tree causes multiple node accesses. If such an access is performed repeatedly, unnecessary expense is incurred. This extra expense is avoided by accessing the node directly with a tree pointer. At the same time, the tree pointer gives the PLANS programmer the opportunity to substitute a brief meaningful name for the cumbersome qualifier list. Many hours of a maintenance programmer's time will be saved if the original programmer pays attention to this type of readibility enhancement. The example below demonstrates the degree of clarity that can be achieved.

```
IF $JOB_NETWORK.JOB_37.SCHEDULE_INTERVAL.START<10
    THEN IF $JOB_NETWORK.JOB_37.SCHEDULE_INTERVAL.END>=15
        THEN GRAFT $JOB_NETWORK.JOB_37.SCHEDULE_INTERVAL AT $WINDOW(NEXT);
```

without a tree pointer;

```
DEFINE $INTERVAL AS $JOB_NETWORK.JOB_37.SCHEDULE_INTERVAL;
IF $INTERVAL.START<10
    THEN IF $INTERVAL.END>15
        THEN GRAFT $INTERVAL AT $WINDOW(NEXT);
```

with a tree pointer.

3.2.2 ADVANCE Statement

The standard form of this statement is

```
ADVANCE tree-pointer-name;
```

The ADVANCE statement allows the PLANS programmer to update tree pointers. It effectively "moves" the pointer one node to the right. That is, after the statement is executed, the pointer will indicate the next node, at the same node level, immediately to the right of the node previously indicated. If there is no such node, the tree pointer will indicate $NULL.

If the node referred to by a tree pointer is pruned or grafted, the pointer is automatically advanced. Because of this property, the ADVANCE statement can be very useful for programming some kinds of explicit loops. For instance, in the example below it is used to conditionally balance the PRUNE statement that causes the pointer to be automatically advanced. By using this technique, the user can insure that all subnodes are examined by the loop.

```
DEFINE $POINTER AS $SAMPLE_TREE(FIRST);
DO WHILE ($POINTER ¬IDENTICAL TO $NULL);
     IF condition
          THEN PRUNE $POINTER;
          ELSE ADVANCE $POINTER;
     END;
```

NOTE: Of course, this particular operation could be accomplished with the single statement: PRUNE $SAMPLE_TREE(ALL: condition);

3.3 ARITHMETIC STATEMENT

3.3.1 Arithmetic Assignment Statement

The arithmetic assignment statement of PLANS, which is of the form (arithmetic variable) = (arithmetic expression), is essentially the same as that of other high-level programming languages.

An arithmetic variable is any character string not more than 30 characters long, where the first character is alphabetic and all others either alphanumeric or the special underbar ("_") character. Arithmetic variables may be of either integer or real types; variables starting with the letters I through N are implicitly declared integer, otherwise they are real. (Note tha. the type declaration is done automatically.) PLANS variables cannot be keywords, so LABEL, NUMBER and so forth are illegal names for arithmetic variables.

Arithmetic expressions are evaluated according to the priority of the operator. Any expression enclosed in parentheses is evaluated before any other part of the expression. Exponentiation (**), prefix + and prefix - have the highest priority. These operations will be completed first, and if more than one of these operators appear in the same expression, they are evaluated from right to left. Multiplication (*) and division (/) have the second priority. They are evaluated from left to right. Addition (+) and subtraction (-) have the lowest priority. They are evaluated from left to right. If any other order is desired, parentheses must be used to indicate the order.

When tree references appear in an arithmetic context, automatic type conversion occurs. For example, the arithmetic statement THIS_IS_AN_ARITHMETIC_VARIABLE = $X(2) + 15 causes the tree reference $X(2) to be converted to the value of the node (say the value of the node is 3) so that the arithmetic statement becomes: THIS_IS_AN_ARITHMETIC_VARIABLE = .3E+01 + 15. Note that $X(2) is automatically converted to floating point. This will be true for any tree reference which appears in an arithmetic context.

Several special function keywords can appear in an arithmetic context. These include the NUMBER function (see section 2.2.10), the LABEL function (see section 2.2.9), and the keyword INFINITY, which refers to a very large number (the exact value is implementation specific). The following are examples of valid arithmetic assignment statements: INTEGER = NUMBER($X) * 3.2 -LABEL($Y(2)), and $X.A(NEXT) = 15.2 - NUMBER($Y).

3.4 CONDITIONAL STATEMENTS AND EXPRESSIONS

3.4.1 IF Statement

The PLANS statement capability includes both IF...THEN and IF...THEN...ELSE constructs. The characteristic make-up of these constructs are:

```
IF Boolean expression
   THEN executable statement;
IF Boolean expression
   THEN executable statement;
   ELSE executable statement;
```

The following are examples of the two types of conditional statements:

```
IF $X(2) + 3 = NUMBER($Y) - 1
   THEN $X = $Y;
```

```
IF NUMBER($X) = 5 & $Y(1) < 4 | LABEL($Z(2)) = 'A'
   THEN GO TO LOOP;
   ELSE STOP;
```

Note that all types of Boolean expression can follow the keyword IF. The different types of Boolean expressions will be described in detail in the following section.

Executable statements are all statements which can be executed. Examples of statements which cannot be executed are: the END statement, Boolean expressions, a PROCEDURE statement, and so forth.

If more than one executable statement must be included in a THEN or ELSE clause, noniterative DO statements (see section 3.7.3) may be used. For example:

```
IF $X(2) < $Y.B
   THEN DO;
      $X(2) = $Y.B;
      LABEL($X(2)) = 'B';
      END;
   ELSE DO;
      IF $X(3) < $Y.C
         THEN $X(3) = $Y.C;
      $Y = $Z;
      TRACE OFF;
      END;
```

IF statements can be nested to any depth, as the following example illustrates:

```
IF $X = $Y
   THEN IF $X.A = 2
```

```
      THEN IF $X.B = $Z(1)
         THEN;
         ELSE TRACE HIGH;
      ELSE;
   ELSE GO TO START;
```

The above example also illustrates the use of null THEN and ELSE clauses, which are used to hold position and clarify nested IF statements. Whether this is an acceptable programming practice is for the reader to decide. With sufficient ingenuity, such null statements can be avoided.

In a nested IF statement, each ELSE clause is associated with the innermost IF...THEN which has no ELSE clause already attached. Thus the following statement,

```
IF $X = $Y THEN IF $Y = $Z THEN TO TO START; ELSE RETURN;
```

is interpreted as:

```
IF $X = $Y
   THEN IF $Y = $Z
      THEN GO TO START;
      ELSE RETURN;
```

To associate the ELSE RETURN clause with the outermost IF statement, a null ELSE clause may be used:

```
IF $X = $Y
   THEN IF $Y = $Z
      THEN GO TO START;
      ELSE;
   ELSE RETURN;
```

or the entire structure can be modified to:

```
IF $X ¬= $Y
   THEN RETURN;
   ELSE IF $Y = $Z
      THEN GO TO START;
```

3.4.2 Boolean Expressions

Boolean expressions can appear in three contexts: in IF-clauses (see section 3.4.1), in conditional tree references ("FIRST:" and "ALL:" see sections 2.2.6 and 2.2.7), and in DO WHILE statements (see section 3.8.1). The following examples illustrate the use of Boolean expressions in the three contexts:

```
IF $X(2) = 15 - $Y.A THEN...ELSE...
$X(FIRST: LABEL($ELEMENT) = 'A')
$Y(ALL: NUMBER($ELEMENT) = 3)
DO WHILE (A > 10)
```

3.4.2.1 Arithmetic Relations

PLANS allows a complete set of standard arithmetic relations. They are:

= (equal to)

¬= (not equal to)

> (greater than)

¬> (not greater than)

>= (greater than or equal to)

¬>= (not greater than or equal to)

< (less than)

¬< (not less than)

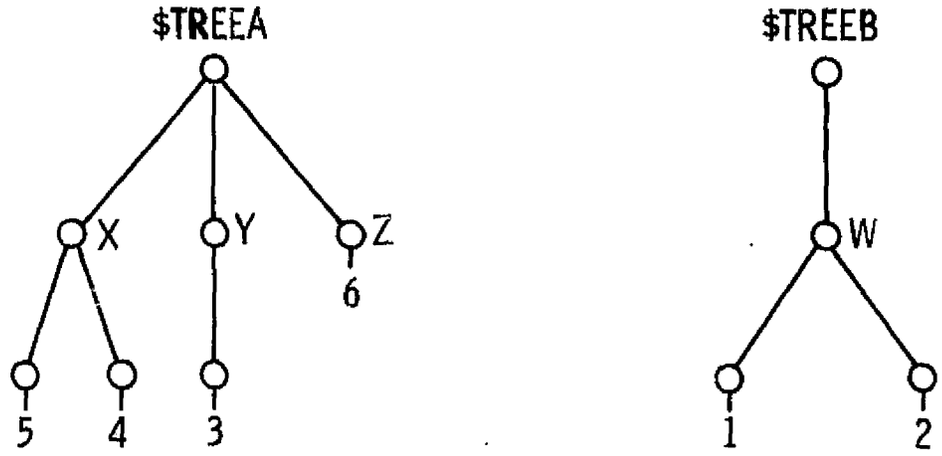
<= (less than or equal to)

¬<= (not less than or equal to)

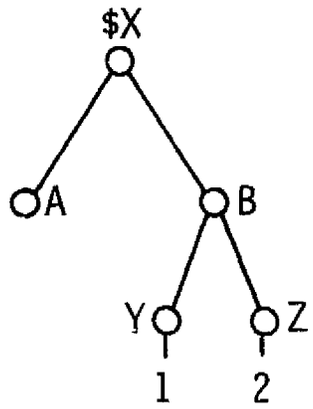
Note that the arithmetic relation "=" compares values, so it should not be used to compare tree structures, or misleadingly true Boolean expressions may result. For example, the two trees in figure 3-21(a) are obviously not the same, yet the Boolean expression $TREEA = $TREEB is true. Since "=" compares values, values are expected on both sides of the "=" sign. However, $TREEA is not a terminal node so it has no value. In this case, the value zero is assigned to $TREEA. The same thing happens to $TREEB: since it has no value, it is automatically assigned the value zero. Thus the value of $TREEA is equal to the value of $TREEB.

The above example illustrates the importance of remembering that arithmetic relations compare values; to compare tree structures, the tree relations described in section 3.4.2.2 should be used (in particular, the tree relation IDENTICAL TO should be used in place of "=").

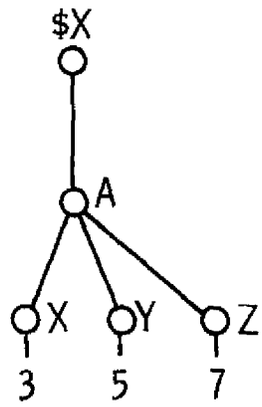
The default value for any tree node which has no value is a function of the arithmetic relation. "=" and "¬=" will automatically assign the value zero or '' (a null character string) to a tree node which has no value, depending on the context. Thus in figures 3-21(b) and (c), the Boolean expressions $X.A = 0 and $X.A = '' are both true; while the expressions $X.A ¬= 0 and $X.A¬= '' are both false. All of the other eight arithmetic inequality relations listed above automatically assign the value zero to any tree node which has no value. Thus, the eight inequality relations are capable of numeric comparisons only, while the "=" and "¬=" relations can do both numeric and character string comparisons.



(a) $TREEA = $TREEB



(b) $X.A =0 , $X.A="



(c) $X.A=0 , $X.A = "

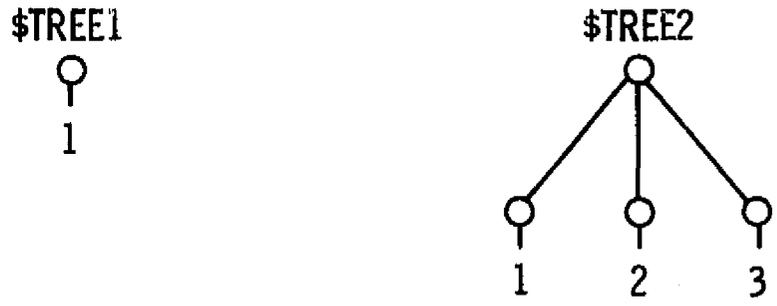
Fig. 3-21 Automatic Evaluation Of Tree Nodes Used With Arithmetic Relations.

3.4.2.2 Tree Relations

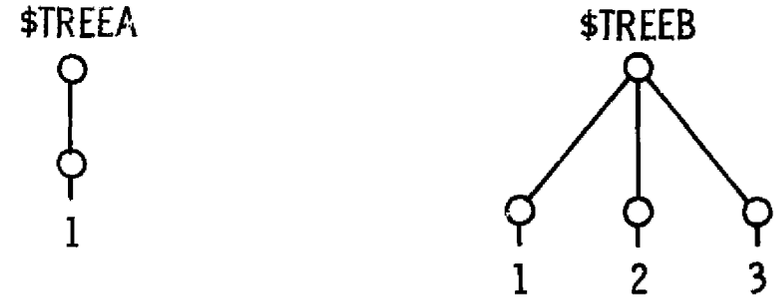
Three PLANS tree relations provide considerable condition-testing power. They are ELEMENT OF, SUBSET OF, and IDENTICAL TO. Examples are illustrated in figure 3-22. Figure (a) illustrates a situation in which the Boolean expression, $TREE1 ELEMENT OF $TREE2, is true. This expression compares $TREE1 with the subnodes of $TREE2. Figure (b) illustrates the expression, $TREEA SUBSET OF $TREEB. Here, the subnodes OF $TREEA are compared with the subnodes of $TREEB. The expression, $TREEX IDENTICAL TO $TREEY, is illustrated in figure (c). This expression tests for complete identity between the two trees.

The ELEMENT OF tree relation can be better understood if one thinks of a tree node as a set of elements, each of which is one of its subnodes. Thus, in figure 3-23(a), the elements of the node $X.A are the nodes labeled X, Y and Z. The Boolean expression illustrated in figure (b), $W ELEMENT OF $X.A, is then really asking, "Is $W an element of $X.A? In other words, does $W appear as a subnode of $X.A?" Figure (c) illustrates a more complicated example of the tree relation ELEMENT OF.

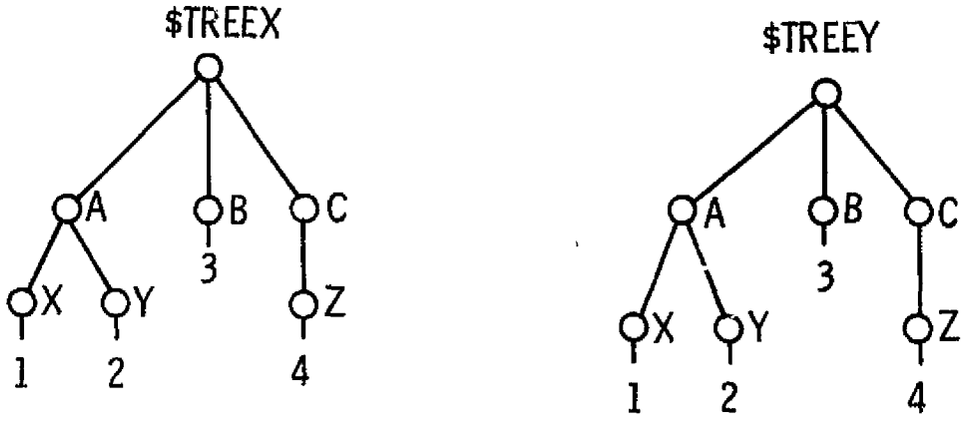
If one continues to think of a tree node as a set of elements (as in figure 3-23 where $X.A was the set consisting of the elements labeled X, Y and Z), then one can easily see that the relation SUBSET OF tests whether each element of one set is also contained in another set. For example, $TREEM.X SUBSET OF $TREEP.Y tests whether the subnodes of $TREEM.X are also subnodes of $TREEP.Y.



(a) $TREE1 ELEMENT OF $TREE2

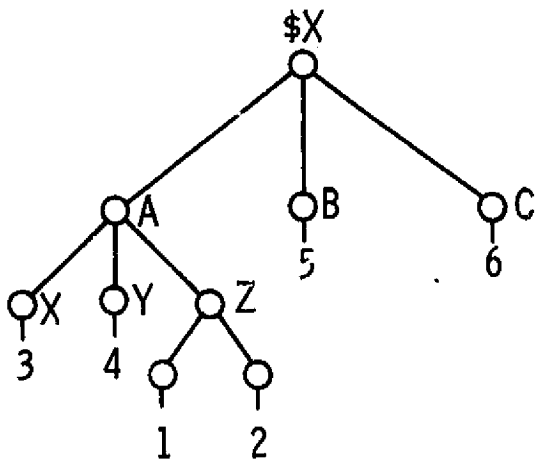


(b) $TREEA SUBSET OF $TREEB

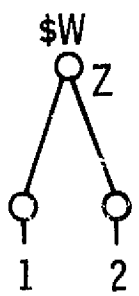


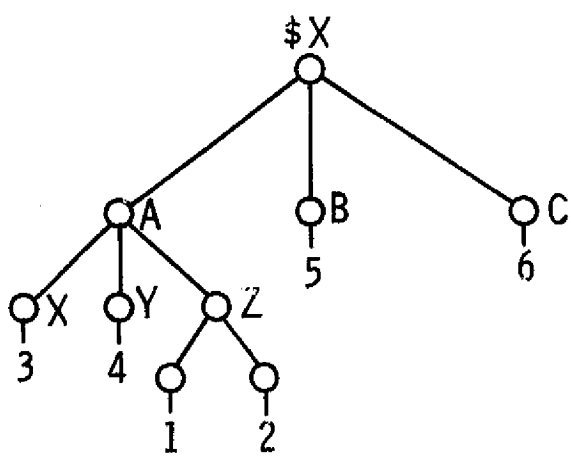
(c) $TREEX IDENTICAL TO $TREEY

Fig. 3-22 PLANS Tree Relations

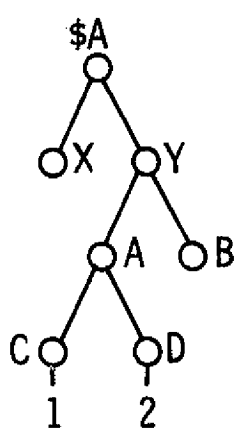


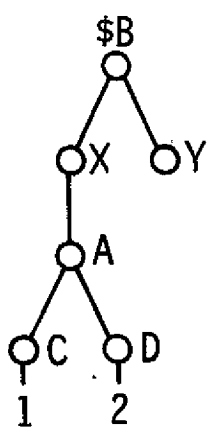
(a) TREE $X





(b) $W ELEMENT OF $X.A





(c) $A.Y(1) ELEMENT OF $B.X

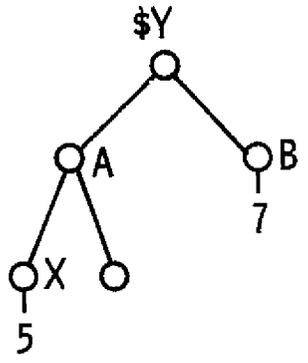
Fig. 3-23 The ELEMENT OF Tree Relation

A particularly common expression in PLANS takes the form $X IDENTICAL TO $NULL, where $NULL is the name of a tree which has no label and no value or substructure (see section 2.2.11). Figure 3-24 (a) illustrates the Boolean expression, $Y.A(2) IDENTICAL TO $NULL. Since $Y.A(2) has no label and no value or substructure, the node is identical to $NULL. Figure (b) illustrates a slightly different case: here, $Y.A(2) does not exist so the node has no label and no value or substructure. In this case, also, $Y.A(2) is identical to $NULL.

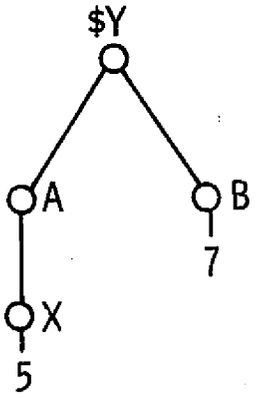
The tree relations ELEMENT OF and IDENTICAL TO may also be used to compare character strings or numeric values with tree structures. Examples are illustrated in figure 3-25. Note that in each example, a dummy node is created (for comparison purposes only) which has, as its value, the character string or numeric value.

The tree relation SUBSET OF cannot be used to compare character strings or numeric values with tree structures, because any Boolean expression of the form, (character string or numeric value) SUBSET OF (tree reference), will always be true. Note that the dummy node created for comparison purposes has no subnodes, so the set represented by the dummy node is empty, and an empty set is always a subset of any set. Thus, a character string or numeric value is always a subset of any tree node.

The keyword "NOT" may be used to negate tree relations only. For example, $TREE1 NOT ELEMENT OF $TREE2, $TREEA NOT SUBSET OF $TREEB, and $TREEX NOT IDENTICAL TO $TREEY are valid statements. Note that $VALUE_A NOT = $VALUE_B and other expressions of this type which combine NOT with an arithmetic operator are not valid.

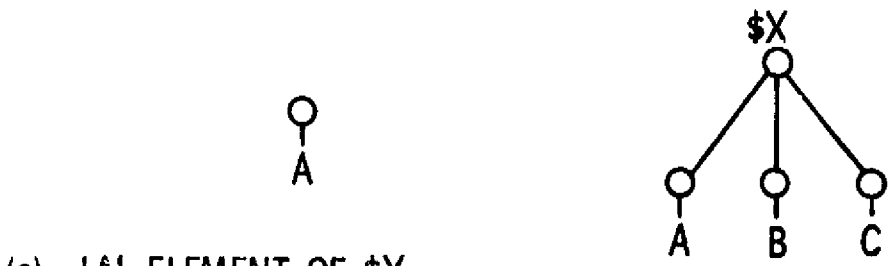


(a) $Y.A(2) IDENTICAL TO $NULL

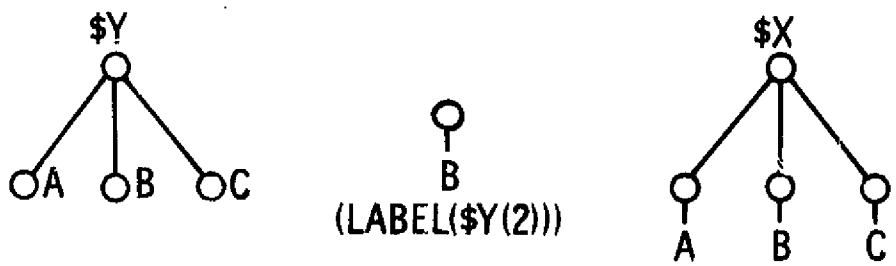


(b) $Y.A(2) IDENTICAL TO $NULL

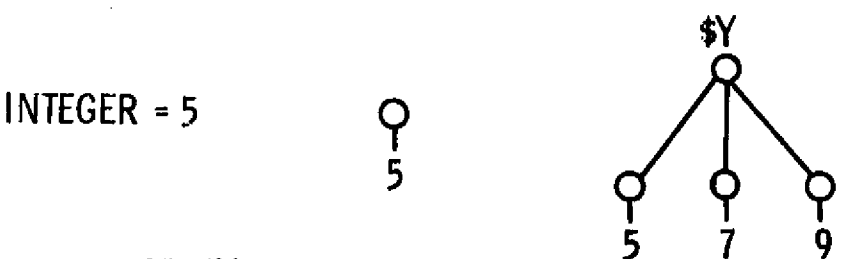
Fig. 3-24 "IDENTICAL TO $NULL"



(a) 'A' ELEMENT OF $X

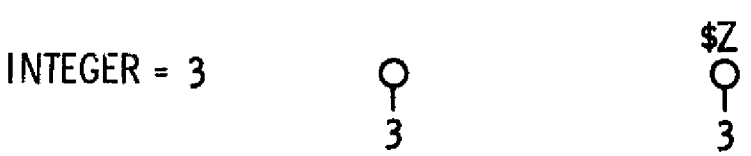


(b) LABEL($Y(2)) ELEMENT OF $X



(c) INTEGER ELEMENT OF $Y

(d) 3 IDENTICAL TO $Z



(e) INTEGER IDENTICAL TO $Z

Fig. 3-25 Tree Relations: Comparisons Between Character String Or Numeric Values And Tree Structures

4.2.3 Logical Operations

There are three logical operations on Boolean expressions which are available in PLANS: "¬" (NOT), "&" (AND) and "|" (OR). These operations take the form:

¬ (Boolean expression)

Boolean expression & Boolean expression

Boolean expression | Boolean expression

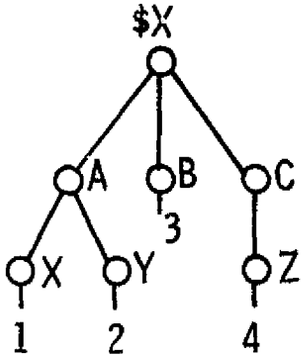
Examples are illustrated in figure 3-26.

The "¬" operator logically negates any simple or complex Boolean expression (see figure (a)). Note that the Boolean expression to be negated is always enclosed in parentheses.

The "&" and "|" operators provide a means for combining simple Boolean expressions. Any Boolean expression of the form, Boolean expression #1 & Boolean expression #2, is true only if both of the Boolean expressions are true (see figure (b)). On the other hand, an expression like Boolean expression #1 | Boolean expression #2 is true if either Boolean expression #1 or Boolean expression #2 is true or both (see figure (c)).

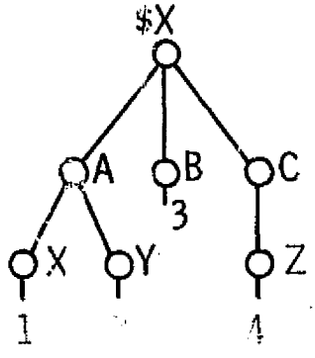
If all three logical operators are found in a Boolean expression, the expression is evaluated as follows: from left to right, with "¬" having the highest priority, followed by "&", then "|". The following example illustrates this point:

$X < $Y & LABEL($X(2)) = 'B' | ¬($Y.A > $Z.C) & $Z(1) = 2.3 is evaluated as (($X < $Y) & (LABEL($X(2)) = 'B')) | ((¬($Y.A > $Z.C)) & ($Z(1) = 2.3)).



(a) ¬ (X. B < 2)

(b) ($X. B = 3) & ($Y(2). Y = 6)



(c) $X(3) . Z = 4 | LABEL ($X(2)) = 'E'

Fig. 3-26 PLANS Logical Operations

3.5 CONTROL AND TRANSFER OF CONTROL STATEMENTS

3.5.1 GO TO Statement

The standard form of this statement is

GO TO statement_label;

An example is GO TO TRY_NEXT_NODE. The GO TO statement causes an unconditional branch to the statement with the specified label. It is clear that this functional capability is desirable. It is often true, however, that the use of appropriate structures (DO-groups, etc.) can eliminate the necessity for GO TO statements while increasing program simplicity and clarity. PLANS allows structured or unstructured programming, at the programmer's option, but lends itself especially well to the former.

GO TO statements in internal procedures may have as their destinations any statement in a containing procedure. However, they cannot be used to branch from a procedure into the middle of one of its internal procedures.

3.5.2 CALL Statement

The standard form is

CALL procedure_name (argument_list);

An example is CALL PROCEDURE_A (A,B,$X). The CALL statement invokes the specified procedure, causing program control to be transferred to it. The statement immediately following the CALL statement automatically receives control when the called procedure finishes executing.

Arguments may be passed to the procedure via a parenthesized list. This argument list may consist of integer variables, real variables, and/or trees. It is the programmer's responsibility to insure that

the data attributes of the calling arguments match those of the corresponding procedure parameters. Failure to do so will result in conversion errors at execution time.

3.5.3 RETURN Statement

The return statement causes return of control from the currently executing procedure to the statement immediately following its invoking CALL statement. RETURN statements are needed to specify conditional returns in the middle of a subprogram. Otherwise, they need not be used, since the final END statement of a procedure causes an automatic return.

3.5.4 STOP Statement

The STOP statement causes an entire program to be aborted. Its use should be reserved for abnormal terminations. Normal program termination occurs when the final END statement of the main procedure is executed, or a RETURN statement is encountered in the main program.

3.5.5 TRACE Statement

```
TRACE trace_level_indicator;
```

The TRACE statement provides a simple mechanism by which the user can cause trace information to be output by his program. This debugging tool can be used to control both the type and frequency of trace output.

If TRACE statements appear in a program, the TRACE option (see section 3.7.1, "PROCEDURE Statement") must be in effect.

It is important to note that the TRACE statement is executable. Therefore, it allows the user to dynamically vary the characteristics of the trace output during program execution. For instance, a program could be set up so that trace information is only output if certain error conditions arise during execution. Based on the conditions, the logic could dynamically select which sections of PLANS code would be traced. This would eliminate irrelevant trace output, directing the programmer's attention to those areas that are more likely to contain the bug.

There are three "trace-level-indicators" (OFF, LOW, and HIGH) used to specify the type of trace information desired. Of course, TRACE OFF, specifies that no trace output is desired. If TRACE LOW is selected, the numbers of all executable statements are output just before their execution. This provides a detailed record of the program logic flow. TRACE HIGH is used to obtain the maximum amount of detailed trace information. This includes statement numbers, all changes to variable values, and all changes to tree nodes.

3.6 INPUT/OUTPUT STATEMENTS

3.6.1 READ Statement

The standard form of this statement is:

READ input_item_list.

The READ statement allows the programmer to input data from a file external to the source program. This file could be on cards, disk, tape, etc. The "input_item_list" can consist of integer or real variables and tree node references.

To read in a value to be assigned to a specified variable, the numeric constant is placed anywhere within the first twenty columns of the input file record. The following are examples of legal numeric constants: 3.1416, -5, .037, 0092, -78.1, +61.43E-01. The last item in this list of sample constants is written in the conventional exponential notation common to many other programming languages.

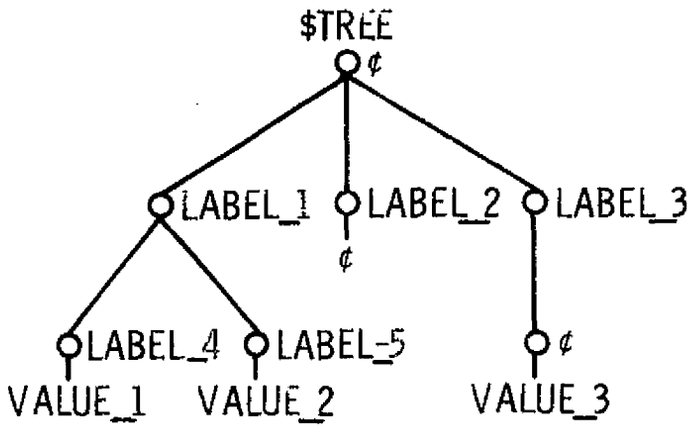
When reading in a tree structure, if the target node does not exist yet, it will be established by the READ statement. Trees are represented, on input, in a standardized indented format. Each card image in the input file represents a tree node and may contain both a label and a value. Figure 3-27 is an example, showing a tree and its corresponding indented textual representation. Note that if a label is indented three spaces more than the label immediately preceding it, the corresponding node is interpreted as a subnode of the preceding node. The maximum length for any label or value is 31 characters. Null labels are represented by a cent sign ("¢"). The input tree is terminated by the keyword "END" beginning in column one.

3.6.2 WRITE Statement

The standard form of this statement is:

WRITE output_item_list.

The WRITE statement is used to output a character string literal, a specified tree node with its value or substructure, or the value of an integer or real variable. Character strings must be enclosed




```
COLUMN 1
    COLUMN 4
        COLUMN 7
¢   ¢   ¢
    LABEL_1
        LABEL_4 - VALUE_1
        LABEL_5 - VALUE_2
    LABEL_2 - ¢
    LABEL_3
        ¢ - VALUE_3
END
```

FIG 3-27 Indented Tree Format

in single quotes (e.g., WRITE 'THIS IS A CHARACTER STRING LITERAL';). Tree structures are output in the same indented format as described for input (see Section 3.6.1).

The values of any variables that appear in the output item list are output in the conventional "E" format. For example if the WRITE statement was used to output a value of -12.345 the following would result: -1.234500E+01.

3.7 STRUCTURAL STATEMENTS

PLANS is a hierarchic block-structured language. Each program block is a separate logical entity and may have its own local storage. The block structure of PLANS is determined by a set of PROCEDURE blocks, BEGIN blocks, and DO groups. The blocks may be nested at will, yielding a hierarchic structure such as that shown in Figure 3-28. Note that each block and group must be terminated with its own END statement.

If the programmer takes advantage of the characteristics of a block-structured language, program structure can be greatly simplified while simultaneously improving program readability. Block structure also tends to increase the power of a language by providing a natural mechanism for treating a whole block of statements as single unit. In a high-level language, such as PLANS, these blocks correspond directly to the logical blocks in terms of which the software designer thinks about his problem.

In PLANS, all tree and variable names are automatically given <u>global</u> scope, unless they are: (1) explicitly declared LOCAL, or (2) procedure parameters. In either of these two cases the names become local to the block with which they are associated. That is, their values or structures are not known outside of the block. Since this convention is just the opposite of that used by most non-structured languages (e.g., FORTRAN, COBOL) it can be difficult to grasp.

Any tree that is not a parameter but is declared LOCAL to a procedure or BEGIN block is automatically pruned on exit from the block. This insures that such trees will be inaccessable to all statements outside of the block, and automatically makes the storage associated with

```
A:   PROCEDURE;
     statement - a1
     statement - a2
     B:   DO I = 1 TO 10;
          statement - b1
          statement - b2
          C:   PROCEDURE;
               statement - c1
               statement - c2
               D:   DO;
                    statement - d1
                    statement - d2
                    E:   BEGIN;
                         statement - e1
                         statement - e2
                         END;
                    statement - d3
                    END;
               statement - c3
               END;
          statement - b3
          END;
     statement - a3
     END;
```

Figure 3-23

PLAMS Hierarchic Block Program Structure

these trees available for reuse. The values of LOCAL variables, likewise, become unavailable on exit from the block.

3.7.1 PROCEDURE Statement

The standard form of this statement is:

label: PROCEDURE(parameter-list) OPTIONS(option-list) RECURSIVE;

In the above format only the label and PROCEDURE keyword must always appear. All other items are optional.

There are three kinds of procedures in PLANS: main, external, and internal. Main and external procedures are compiled independently, while internal procedures are nested inside other procedures. Every PLANS main program must begin with a MAIN procedure statement and end with the closing END statement. In the execution of any PLANS program the main procedure always receives initial control. It may then transfer control to a subprogram written as an internal or external procedure. This can only be done by means of a CALL that refers to the label of the appropriate procedure. Thus, in figure 3-28, if statement-b2 is not a CALL or other transfer-of-control statement, it will be followed logically by statement-b3, with transfer of control skipping around internal procedure C.

When using an internal or external procedure as a subprogram, it is usually desirable to pass arguments as a part of the CALL statement. The calling argument list corresponds directly to the parameter list specified in the PROCEDURE statement. The variable and/or tree names

appearing in the parameter list automatically assume local scope with respect to the called procedure. The called procedure uses these names to refer directly to particular arguments passed to it by the invoking CALL statement.

Internal or external procedures may also be declared RECURSIVE by specifying this keyword on the PROCEDURE statement. This must be done if a procedure: (1) directly calls itself, or (2) indirectly calls itself by invoking another procedure that causes it to be called. RECURSIVE procedures can be extremely powerful tools, although programmers unfamiliar with their use may find it difficult to take advantage of the added capability they provide. Their operation is relatively sophisticated and users are well advised to read carefully the following explanation. Each time a procedure is invoked recursively, all of its local storage is reallocated, and the previous allocation is pushed down in a stack. Each time a recursive execution of a procedure is terminated, local storage is popped up, yielding the next most recent generation of local storage.

No global variables or trees are saved on the stack. Operation of the stack provides a mechanism for preserving each generation of local storage. This makes it possible to restore the execution environment associated with each recursive call.

A main or external procedure statement can specify a list of translation-time options. Most of the options occur in pairs, specifying that a certain option was either "on" or "off". In the list below, the default

options are underlined. Each column except the last represents a mutually exclusive pair of options.

MAIN	NOTES	STAT	TRACE	NODES (400,800)
EXTERNAL	NONOTES	NOSTAT	NOTRACE	

The MAIN and EXTERNAL options specify the procedure type. An "internal" option is not needed since this is always obvious from the program structure. The NONOTES option supresses output of all diagnostic messages classified as notes. The STAT and TRACE options are very useful but will cause an increase in program execution time and storage. The STAT option causes generation of statistic-keeping code. Similarly the TRACE option causes generation of code that will provide an execution-time trace very useful for debugging purposes. Note that this option must be selected if any trace statements (see section 3.5.5) appear in the program. The NODES option can only be used if the MAIN option is also in effect. It is used to specify: (1) the number of tree node storage spaces to be reserved, and (2) the number of 8-character blocks to be reserved for tree node labels and values. Since all labels and values are stored as character strings, space for them is allocated in the same way by the PLANS "buddy" system dynamic storage allocation routine. If a program exceeds the default label-value storage space allocation, the table below can be used to estimate the value to be used when overriding it with the NODES option.

NUMBER OF CHARACTERS IN LABEL OR VALUE	NUMBER OF 8-CHARACTER STORAGE BLOCKS REQUIRED
1 through 7	1
8 through 15	2
16 through 31	4

3.7.2 DECLARE Statement

The standard form of this statement is:

DECLARE declare-item-list LOCAL;

In PLANS, DECLARE statements are used only to control variable scope and not to declare data types. Data types are implicitly declared by the first character of the identifier. All identifiers beginning with a dollar sign ("$") are assumed to be trees or tree pointers. All identifiers beginning with one of the letters I through N are assumed to be integer variables. All identifiers beginning with any other alphabetic character are assumed to be real variables.

DECLARE statements, if they occur, must immediately follow the appropriate internal procedure statement or BEGIN statement. Any variables, trees, or tree pointers used in the program block may appear in the "declare-item-list." However, procedure parameters are automatically assumed to be local. Any trees that are explicitly declared LOCAL (except procedure parameters) will automatically be pruned on exit from the block. This guarantees that the storage space taken up by these trees will be released for reuse.

LOCAL declarations are never needed in a MAIN or EXTERNAL procedure, since in this context local and global have the same meaning. In the use of an EXTERNAL procedure, all trees (except parameters) will automatically be pruned on exit.

3.7.3 Noniterative DO Statement

The form of this statement is:

```
DO;
```

It is often convenient to be able to group several statements together into a single logical unit. The noniterative DO statement provides this capability. It can often be on the THEN or ELSE clause of an IF statement (see section 3.4.1) to maintain sequential control. The example below demonstrates the use of this statement to avoid unnecessary GO TO statements.

Coded with GO TO statements:

```
        IF VALUE < 5
            THEN GO TO LABEL_1;
        PRUNE $TREE;
        WRITE 'MESSAGE2';
        GO TO LABEL_2;
    LABEL_1: GRAFT $TREE AT $SAVE(NEXT);
        WRITE 'MESSAGE1';
    LABEL_2:
```

Coded with Noniterative DO statements:

```
        IF VALUE < 5
            THEN DO;
                GRAFT $TREE AT $SAVE(NEXT);
                WRITE 'MESSAGE1';
                END;
            ELSE DO;
                PRUNE $TREE;
                WRITE 'MESSAGE2';
                END;
```

3.7.4 BEGIN Statement

The standard form of this statement is:

```
BEGIN;
```

This statement is used for the same purpose as the noniterative DO statement, but it also allows the declaration of local variables and trees. The BEGIN statement should not be used unless this additional capability is actually required.

3.7.5 END Statement

The form of this statement is:

```
END;
```

The END statement is used to terminate any PROCEDURE block, BEGIN block, noniterative or iterative DO group. Every block or group must have its own associated END statement.

3.8 ITERATION STATEMENTS

3.8.1 DO WHILE Statement

The DO WHILE statement allows a statement or group of statements to be executed repeatedly as long as a specified condition is true. For example:

```
DO WHILE (I <5);
   $X(I) = 2;
   I = I + 1;
   END;
```

Like all iterative DO statements in PLANS, the group is not executed at all if the condition is found not to be satisfied the first time. Note that the condition is evaluated before execution of the loop, rather than after. For example, the following DO WHILE group will not be executed at all:

```
   .
   .
   .
   K = 3;
   DO WHILE (K > 5);
      $Y(K) = $X(K) + 1;
      LABEL($Y(K)) = LABEL($X(K));
      K = K - 1;
      END;
```

If it is desired to execute a loop until a condition is satisfied, DO WHILE is used with the Boolean expression negated. For example, the following group of statements will be executed until $Y(I) is identical to $NULL:

```
.
.
.
I = 1;
DO WHILE (¬($Y(I) IDENTICAL TO $NULL));
   GRAFT $Y(I) AT $X(I);
   I = I + 1;
   END;
```

3.8.2 Incremental DO Statement

The incremental DO loop allows a group of statements to be executed for each of a specified set of values of a particular variable. The basic structure of the incremental DO loop is illustrated below:

```
DO (variable) = (arithmetic expression),...,(arithmetic expression)
   optional  { TO (arithmetic expression) BY (arithmetic expression)
   elements  {  (arithmetic expression),...,(arithmetic expression)
             { WHILE (Boolean expression);
statement;
statement;
.
.
.
statement;
END;
```

The two basic types of incremental DO loops are:

```
DO (variable) = (arithmetic expression),...,(arithmetic expression);
```

Example: DO I = 5, 3, 7, NUMBER($X)+4, 7-$Y.B;

Explanation: The group of statements will be executed five times; in the five iterations, I will successively assume the values 5,

3, 7, NUMBER ($X)+4, and 7-$Y.B.

DO (variable) = (arithmetic expression) TO (arithmetic expression) BY (arithmetic expression);

Example: DO COUNTER = 5 TO 14 BY 3;

Explanation: The group of statements will be executed four times; in the four iterations, COUNTER will successively assume the values 5, 8, 11 and 14.

Remarks: If the BY-clause is eliminated, the variable is automatically incremented by 1. Thus the statement, DO I = 3 TO 7, will result in five executions of the incremental DO group.

A negative number in the BY-clause will cause the counter variable to be decremented, rather than incremented, by the specified amount. For example, DO VARIABLE = 12 TO 9 BY -2 successively assigns the values 12 and 10 to VARIABLE.

The amount to be incremented or decremented which is specified in the BY-clause need not be an integer. Thus the statement DO X = 6 TO 4 BY -.5 will assign the values 6, 5.5, 5, 4.5 and 4 to X.

Note that the increment condition is tested at the beginning of the DO-group, rather than at the end, so that a DO-group starting DO I = 1 TO 0; would not be executed at all.

The two incremental DO statements described above may be expanded by including any of the optional elements listed in the basic structure. The WHILE-clause, for example, terminates the DO loop whenever a specified Boolean condition is no longer satisfied.

DO (variable) = (arithmetic expression),...,(arithmetic expression) WHILE (Boolean expression);

Example: DO I = 4, 2, 7 WHILE ($X IDENTICAL TO $Y);

Explanation: The DO group will be executed three times or until $X is no longer identical to $Y (whichever occurs first). In the three iterations, I will successively assume the values 4, 2 and 7.

DO (variable) = (arithmetic expression) TO (arithmetic expression) BY (arithmetic expression) WHILE (Boolean expression);

Example: DO VALUE = 10 TO 1 BY -2 WHILE ($X(3) = 5);

Explanation: The DO group will be executed five times or until $X(3) is no longer equal to 5.

Note that the condition is tested at the beginning of the DO-group, rather than at the end, so that a DO-group starting:

A = 20;

DO I = 1 TO 10 WHILE (A < 10); would not be executed at all.

If it is desired to execute a loop until a condition is satisfied, WHILE is used with the Boolean expression negated. For example, the following DO-loop will be executed three times or until $X.A = $Y.B:

DO K = 1, 3, 5 WHILE (¬($X.A = $Y.B));

An example of an incremental DO-loop which utilizes all of the possible options is:

DO COUNT = 2, NUMBER($X), $Y.B, 6 TO $Z(1), 11, 12 TO 8 BY -3, 15 WHILE (¬($Z IDENTICAL TO $NULL));

As long as $Z is not null, the DO loop will be executed and COUNT will successively assume the values 2, NUMBER ($X), $Y.B, 6, 7, 8, ..., $Z(1), 11, 12, 9 and 15.

The variable to which various values are assigned on different executions of the loop is available for use in the loop but ordinarily should not be changed to a different value. The following example illustrates a valid use of the increment variable:

```
DO I = 1 TO NUMBER($X);
$X(I) = I + 3;
LABEL($X(I)) = LABEL($Y(I+1));
END;
```

A possible consequence of changing the increment variable to a different value within the loop is illustrated in the following example:

```
DO I = 1 TO NUMBER($X);
    .
    .
    .
    I = 1;
    END;
```

The statement, I = 1, will cause the DO loop to be executed forever if $X has more than one subnode.

Note that the seemingly identical statements, DO I = 1 TO NUMBER($X) and DO FOR ALL SUBNODES OF $X USING $SUBNODE, do not necessarily cause the loop to be executed an identical number of times. In DO I = 1 TO NUMBER($X), NUMBER($X) is evaluated before the first iteration. Thus, if within the loop another subnode is placed on the tree $X (so that $X has four rather than three subnodes), the loop will still be executed only three times. Depending upon where the new node is placed in the tree $X, the DO FOR ALL SUBNODES loop will be executed three or four times (see section 3.8.3).

3.8.3 DO FOR ALL SUBNODES Statement

The DO FOR ALL SUBNODES loop allows a group of statements to be executed for each of the subnodes of a particular node. In the first iteration of the loop, a tree pointer (see section 3.2) is placed at the leftmost subnode of the specified node. The tree pointer is moved from left to right in successive iterations; in each iteration the pointer points to a different subnode until all of the subnodes have been exhausted.

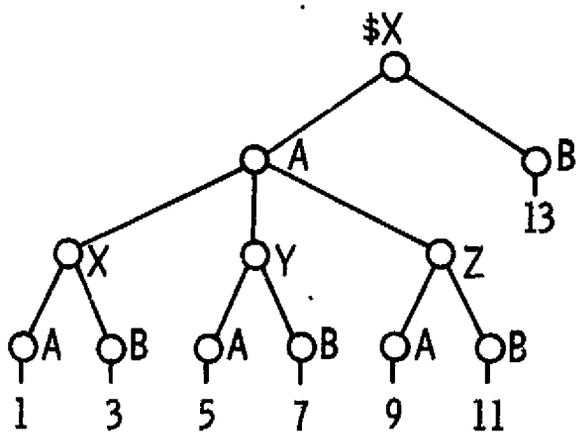
The basic structure of the DO FOR ALL SUBNODES loop is illustrated below:

```
DO FOR ALL SUBNODES OF tree-node-reference USING tree-pointer;
    statement;
    statement;
    .
    .
    .
    statement;
    END;
```

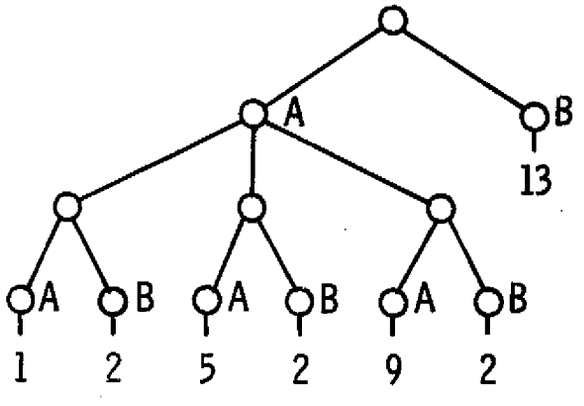
The following example refers to the tree illustrated in figure 3-29:

```
DO FOR ALL SUBNODES OF $X.A USING $A_DESCENDANT;
    LABEL($A_DESCENDANT) = '';
    $A_DESCENDANT.B = 2;
    END:
```

Note that the tree pointer $A_DESCENDANT is available for use within the DO loop, and that it can be qualified by label or subscript like any other tree node reference. However, it must be kept in mind that $A_DESCENDANT.B = 2 really means that $X.A(1).B = 2, $X.A(2).B = 2, and so forth.



(a) Initial Tree



(b) Tree After DO FOR ALL SUBNODES OF $X.A
USING $A_DESCENDANT;
LABEL($A_DESCENDANT)='';
$A_DESCENDANT. B =2;
END;

Fig 3-29 DO FOR ALL SUBNODES Loop

As explained in section 3.2.1, the data attributes of a tree pointer are different from those of a "real" tree, so any attempt to use the same tree name for both purposes will cause an error. For example, the statements:

```
.
.
.
READ $X;
DO FOR ALL SUBNODES OF $TREE USING $X;
     .
     .
     .
```

will cause an error since $X is first used as the name of a tree, and then defined in DO FOR ALL SUBNODES as a tree pointer.

Note, however, that a tree pointer may be used outside of a loop to refer to the last subnode the pointer was identified with within the loop. For example, suppose $X has three subnodes labeled A, B, and C. During the second iteration of the loop, DO FOR ALL SUBNODES of $X USING $X_DESCENDANT, if a branch to a statement outside of the loop occurs, then $X_DESCENDANT will continue to refer to the node $X.B. Thereafter in the program, any reference to $X_DESCENDANT will automatically be a reference to $X.B (as long as $X_DESCENDANT is not redefined). Note that if the branch had not occurred during the second iteration, after the third iteration of the loop $X_DESCENDANT will be pointing to a null node. $X_DESCENDANT does not refer to $X.C <u>after</u> three iterations because the pointer is advanced to the next node (which is null since $X has only three subnodes) before the loop is terminated.

The use of tree pointers outside of a loop is also illustrated in the following example:

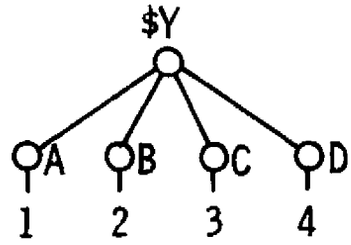
```
$Y = $X(FIRST:NUMBER($ELEMENT)=0);
WRITE $ELEMENT;
```

In this example, $ELEMENT refers to the subnode of $X which has no substructure (i.e., NUMBER($ELEMENT) = 0). In figure 3-29, $ELEMENT thus refers to the node $X.B. Thereafter in the program, any reference to $ELEMENT will automatically be a reference to $X.B (unless $ELEMENT is redefined), so the statement WRITE $ELEMENT will cause $X.B to be written out.

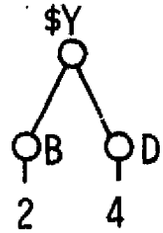
If the tree node reference in the DO FOR ALL SUBNODES statement has no subnodes, the DO loop is never executed. The following is an example of a loop which will not be executed: (the tree referred to is illustrated in Figure 3-29(a))

```
DO FOR ALL SUBNODES OF $X.B USING $B_DESCENDANT;
     statement;
     statement;
     .
     .
     .
     statement;
     END;
```

If the node to which the pointer currently refers is removed from its tree, the pointer will immediately point to the next node. An example is illustrated in Figure 3-30. In the first iteration, $SUBNODE points to $Y.A. The instruction, PRUNE $SUBNODE, removes $Y.A from the tree so $SUBNODE now refers to $Y.B. The END statement



(a) Initial Tree



(b) Tree After
```
DO FOR ALL SUBNODES OF $Y
USING $Y_DESCENDANT;
PRUNE $Y_DESCENDANT;
END;
```

Fig. 3-30 Pruning The Pointer In A DO FOR ALL SUBNODES Loop

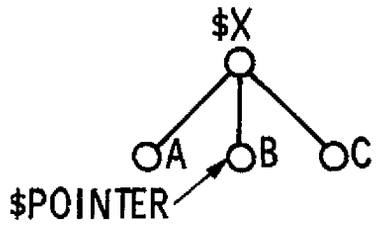
indicates the first iteration is completed, so the pointer is advanced to the next subnode and the second iteration begins. Note that $SUBNODE now refers to $Y.C. $Y.C. is removed from the tree by the instruction, PRUNE $SUBNODE, and the pointer $SUBNODE now refers to $Y.D. The END statement terminates the second iteration, and an attempt is made to advance the pointer. But $SUBNODE is already pointing to the last subnode of $Y, so it can be advanced no further. Thus, there are no other iterations of the DO loop.

If new nodes are added to the left of the node to which the pointer currently refers, they will be ignored; if they are added to the right of the current node, they will be included in the iteration. Examples are illustrated in figure 3-31.

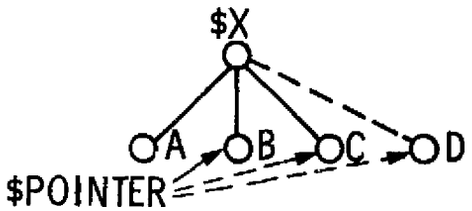
3.8.4 DO FOR ALL COMBINATIONS Statement

The DO FOR ALL COMBINATIONS loop allows a group of statements to be executed for each possible combination (of a specified size) of subnodes of the indicated base node. The indicated base node is considered to be a set whose elements are its subnodes. The DO FOR ALL COMBINATIONS loop generates, one at a time, all combinations of those elements for a given combination size. Thus, for example, one can write DO FOR ALL COMBINATIONS OF $X TAKEN 2 AT A TIME; with the result that the DO loop will be executed once for each 2-element combination of the subnodes of $X.

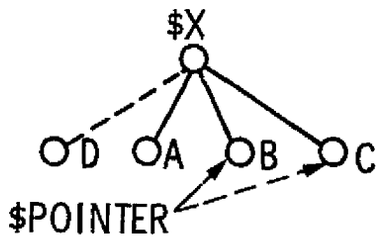
The particular combination that is relevant during an iteration of the DO loop may be referred to within the loop by the reserved tree name $COMBINATION. $COMBINATION should be used by the programmer as if it is the name of a tree which consists of the nodes (and their substructures) making up the present combination. For example,



(a) Initial Tree With $POINTER Pointing To $X.B
$POINTER Is Being Automatically Advanced Each Time
The End Of A DO FOR ALL SUBNODES Loop Is Reached.



(b) If A New Node Is Added At $X(NEXT),
$POINTER Will Advance To C, Then D On
Successive Iterations Of The Loop



(c) If The New Node Is Inserted Before $X(FIRST),
The Position Of $POINTER Is Not Affected,
And $POINTER Will Advance To C Only.

Fig 3-31 Addition Of New Subnodes Within A DO FOR ALL SUBNODES Loop

in figure 3-32, $X has three elements with labels A, B, and C. The combinations of these elements taken two at a time are {A,B}, {A,C}, and {B,C}. As illustrated in figure 3-32, $COMBINATION consists of the nodes $X.A and $X.B during the first iteration of the DO loop, $X.A and $X.C during iteration two, and $X.B and $X.C during the final iteration.

$COMBINATION is not actually a separate tree, but is instead merely a set of tree pointers. This is quite efficient, but results in an implementation restriction which requires that reference be made only to individual subnodes (elements) of $COMBINATION, and then only by subscript reference. Thus, any reference to the tree $COMBINATION is illegal (without subscript qualification); $COMBINATION(1), $COMBINATION(2), and so forth are examples of the only way $COMBINATION may be used in a tree node reference. The reference may be further qualified, however, as in $COMBINATION(3).WEIGHT, $COMBINATION(I).PREDECESSOR(3), etc.

The DO FOR ALL COMBINATIONS loop generates combinations in a standard order. For example, if $Y is a tree with five subnodes labeled A, B, C, D and E, the statement DO FOR ALL COMBINATIONS OF $Y TAKEN 3 AT A TIME would generate combinations in the following order:

A B C

A B D

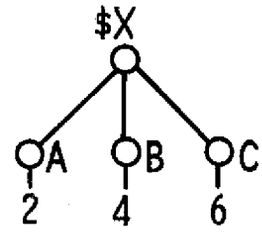
A B E

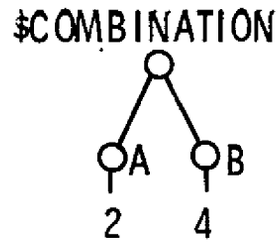
A C D

A C E

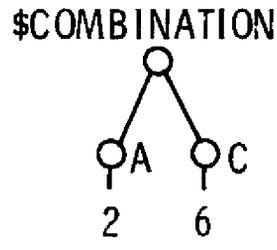
A D E

(a) Original Tree

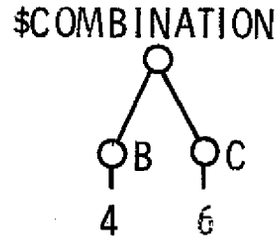




(b) First Iteration Of DO FOR ALL COMBINATIONS OF $X TAKEN 2 AT A TIME



(c) Second Iteration



(d) Third Iteration

Fig. 3-32 Automatic Generation Of Combinations

B C D

B C E

B D E

C D E

Note that if no combinations of the given size exist, the loop is not executed. For example, the statement DO FOR ALL SUBNODES OF $X TAKEN 4 AT A TIME (where $X is illustrated in figure 3-32) would not be executed since $X has only three subnodes.

3.8.5 DO FOR ALL PERMUTATIONS Statement

This statement behaves in the same way as DO FOR ALL COMBINATIONS (see section 3.8.4), except that: (1) all permutations of a given size are generated, and (2) reference is made to $PERMUTATION, rather than $COMBINATION.

4.0 SAMPLE PROGRAMS

4.1 ORDERING OF A PRECEDENCE NETWORK

4.1.1 Problem Statement

This example will assist the reader in gaining a greater intuitive feeling for PLANS dynamic tree operations. The program is called ORDER_BY_PREDECESSORS and its function is to technologically order the list of jobs passed to it in $JOBLIST. A technological ordering requires that any job will appear in the list only after all of its associated predecessor jobs.

4.1.2 Problem Model

In order to illustrate the operation of ORDER_BY_PREDECESSORS on a simple data case, a job network containing four jobs is provided in the input tree, $JOBLIST. The initial state of $JOBLIST is shown in Stage 1 of Figure 4-1. Note that the tree structure provides a natural way of associating each job with its corresponding predecessor jobs.

4.1.3 Program Logic

Implementation of a program to perform this function, while fairly difficult in most programming languages, is very simple and straightforward in PLANS. While there are many functionally equivalent ways to write this program, one of the simplest and most efficient is as follows:

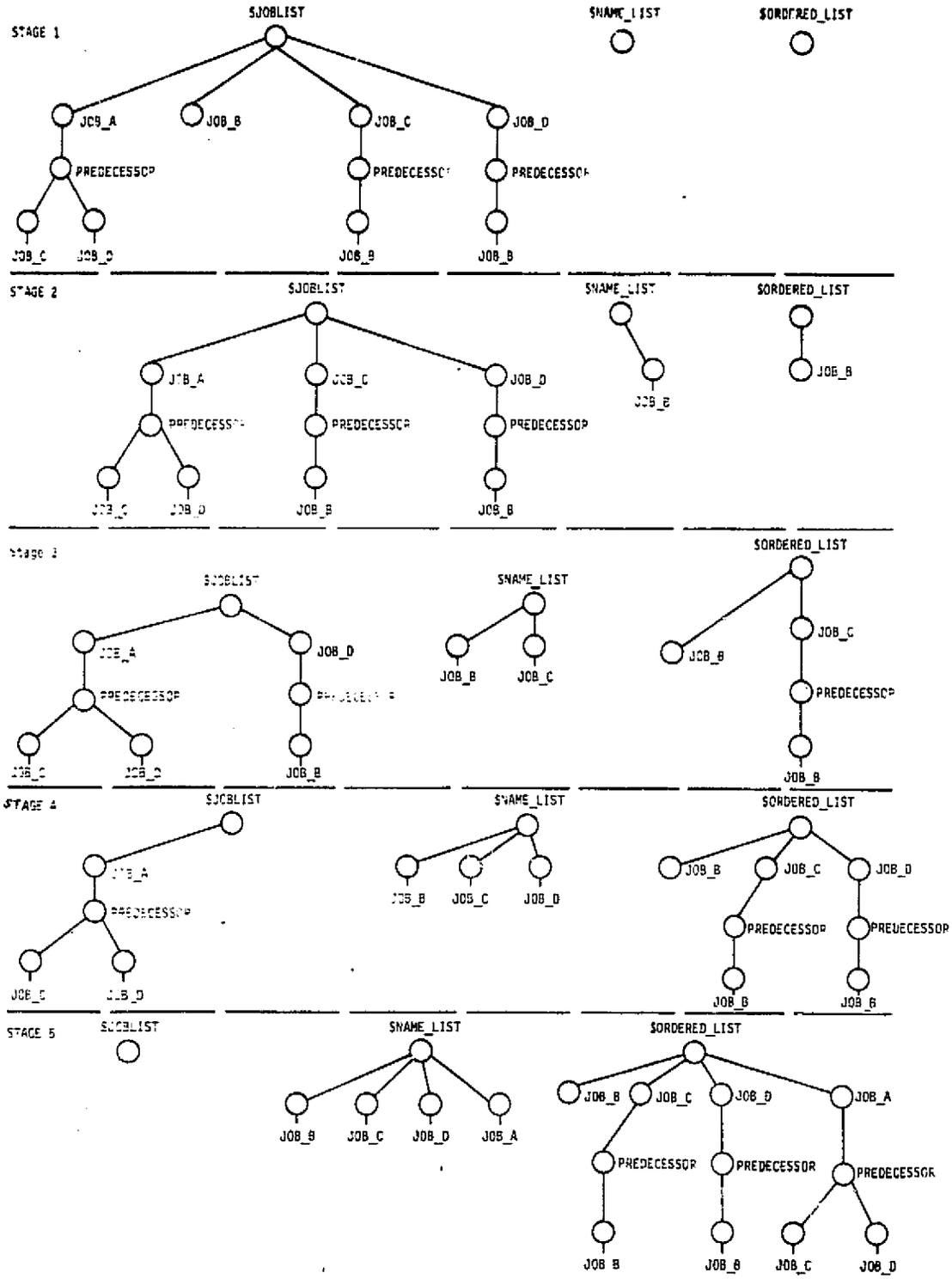


Fig. 4-1 Data Structures Illustrating the Operation of ORDER_BY_PREDECESSORS

```
1  ORDER_BY_PREDECESSORS:  PROCEDURE ($JOBLIST, $ORDERED_LIST) ;
   /*  THIS PROCEDURE TECHNOLOGICALLY ORDERS THE SET OF JOBS INPUT TO    */
   /*  IT IN $JOBLIST AND RETURNS THEM IN $ORDERED_LIST.                 */
2     DECLARE $NAMELIST, $TEMP LOCAL;
3     DO WHILE($JOBLIST(FIRST) NOT IDENTICAL TO $NULL) ;
4        GRAFT $JOBLIST(FIRST:$ELEMENT.PREDECESSOR SUBSET OF $NAME_LIST)
                                          AT $TEMP;
5        IF $TEMP IDENTICAL TO $NULL  THEN RETURN ;
6        $NAME_LIST(NEXT) = LABEL($TEMP) ;
7        GRAFT $TEMP AT $ORDERED_LIST(NEXT) ;
8        END ;
9  END ORDER_BY_PREDECESSORS ;
```

Notice, first (statement 1), that the program is written as an internal procedure with explicit parameters $JOBLIST and $ORDERED_LIST. The calling program will initialize these trees with the structures of the trees specified in the argument-list of the CALL statement.

In statement 2, $TEMP and $NAME_LIST are declared to be LOCAL trees. This means two things: (1) any use of these tree names within this procedure is entirely local, and will not affect trees of the same name outside this procedure, and (2) each time ORDER_BY_PREDECESSORS is called, $TEMP and $NAME_LIST will be initially null, and any storage they use will be made available for reuse upon return without any other action on the programmer's part.

Rather than reordering $JOBLIST, ORDER_BY_PREDECESSORS will iteratively move the jobs, one at a time, from $JOBLIST to $ORDERED_LIST; so that the $ORDERED_LIST will become a correct ordering of the jobs that were originally in $JOBLIST. This

iteration is accomplished via the DO-WHILE statement. The WHILE condition causes the program to check to see if there are any jobs left in $JOBLIST before beginning each successive pass through the loop. When this condition is not satisfied execution of the loop terminates and control is passed to the procedure END statement (statement 9), since it immediately follows the DO group END statement. This terminates the procedure and the complete $ORDERED_LIST is returned to the calling program.

$JOBLIST, on the other hand, will be returned null, assuming all goes well. Upon return from ORDER_BY_PREDECESSORS, then, the calling program will use $ORDERED_LIST where $JOBLIST was used before (or will GRAFT $ORDERED_LIST AT $JOBLIST) after checking $JOBLIST for a null condition.

Note that the data input in $JOBLIST describes a predecessor network in which JOB_B has no predecessor jobs, jobs C and D must each be preceded by JOB_B, and JOB_A must follow both C and D. The diagram shows only information essential for present purposes. However, it is assumed that other information about each job (e.g., duration, resource requirements, etc.) is also present. Because we can access predecessor information by label without regard for its ordinal position, any other information about these jobs is irrelevant, so long as the label PREDECESSOR is used only with the meaning assumed here. $ORDERED_LIST is assumed to be null. Ordinarily this condition

will be assured by the calling program. $TEMP and $NAME_LIST are automatically initialized to a null condition.

Consider now the effect of the GRAFT statement (statement 4) on these trees. This statement specifies that a particular job is to be removed from $JOBLIST and placed at $TEMP. The job to be selected is to be the first job whose predecessor set is a subset of $NAME_LIST. $NAME_LIST will be used to collect the names of the jobs in $ORDERED_LIST, so that the SUBSET OF relation can be used to automatically determine whether the predecessor requirement of a particular job is satisfied. Because $NAME_LIST is presently null, the only job of $JOBLIST that can possibly satisfy the conditional access is a job that has no predecessors. Note that JOB_B fulfills this requirement, and that it is not necessary in this case that a node labeled PREDECESSOR even appear under JOB_B because a nonexistent node has all the properties of a null node, including null subnode structure. JOB_B therefore satisfies the conditional access, and is removed from $JOBLIST and placed at $TEMP. Note that this causes the previously null root node of $TEMP to be replaced by the JOB_B base node and any associated substructure.

Statement 5 now tests for failure of the previous GRAFT statement. In the event that no subnode of $JOBLIST satisfied the access condition, $TEMP will now be null, and detection of this condition can be used to trigger return from ORDER_BY_

PREDECESSORS. However, this could only occur if there were a precedence cycle or a missing job in $JOBLIST. If one of these error conditions is detected, a non-null $JOBLIST will be returned to the calling program, warning it of the problem. In the present case, however, $TEMP is not null. A node is defined as null only if it either does not exist or has both a null value and a null label. Regardless of any substructure, the node $TEMP now has the label "JOB_B" and is therefore not null, so no return occurs.

Statement 6 is therefore executed, placing the name of the job that was found into $NAME_LIST. Several things should be noticed here. Since $NAME_LIST is currently null, $NAME_LIST (NEXT) is equivalent to $NAME_LIST(1). LABEL($TEMP) is a character string. Therefore, a dummy node is established, with a null label, and placed at $NAME_LIST(NEXT). The statement causes the job name, "JOB_B" to be a value of $"NAME_LIST(1), so that subsequent comparisons of $NAME_LIST and PREDECESSOR nodes will find job names as values in both places.

Finally, line 7 is executed, moving the found job, with all descriptive information, from $TEMP to the next available position in $ORDERED_LIST, resulting in the state shown in Stage 2 of Figure 4-1. Note that $TEMP again reverts to a null condition. Trees always have root nodes, although they may be null. Thus, the removal of the node labeled "JOB_B" causes another null node to be placed at $TEMP.

The program has now found the first job that can be executed and has moved it into $ORDERED_LIST. Since the END statement of the DO group (statement 8) has been reached another iteration is initiated (line 8) by jumping back to the beginning of the loop (statement 3).

Stage 3 of Figure 4-1 shows the results of the tree manipulations that occur during iteration 2. The initial state is the same as that shown in Stage 2 of the diagram. The conditional GRAFT statement again searches for a job whose predecessors, if any, are all named in $NAME_LIST. Since $NAME_LIST now contains JOB_B, either a job with no predecessors or a job with only the predecessor JOB_B will satisfy the access condition. The first such job now in $JOBLIST is JOB_C, which is therefore grafted at $TEMP. Because $TEMP is not null, no return is made by statement 5.

Statement 6 places the name of the found job at the next available subnode of $NAME_LIST. As shown in the diagram $NAME_LIST now contains the names of the two jobs (JOB_B and JOB_C) found so far. $TEMP is grafted (statement 7) onto the next available position of $ORDERED_LIST, which now contains all the information about jobs JOB_B and JOB_C (in that order) that was originally in $JOBLIST. Only the jobs not yet placed in $ORDERED_LIST still remain in $JOBLIST. Statement 8 then causes another iteration to begin.

This process is repeated two more times, once for JOB_D and once for JOB_A, with the results shown in Stages 4 and 5 of Figure 4-1, respectively. All jobs have now been moved to $ORDERED_LIST. Since $JOBLIST(FIRST) is now null, the condition tested in statement 3 will not be satisfied and execution of the loop (and of the procedure) will be terminated. $JOBLIST and

$ORDERED_LIST will be returned to the calling program, while $TEMP and $NAME_LIST will be pruned automatically in order to free their storage.

It may occur to the reader to question the use of $TEMP, because a found job could be grafted (statement 4) directly at $ORDERED_LIST(NEXT). However, this would require two additional accesses to $ORDERED_LIST(LAST), one to test for a null condition (statement 5) and one to extract the label (statement 6). In addition, before exit, the extra null node that would have been grafted onto $ORDERED_LIST would have to be removed. It should always be borne in mind that node access time is a function of the number of subnodes that must be scanned (left to right) before the desired node is found. Thus, $TEMP is more efficient to access than is $ORDERED_LIST(LAST), and the difference is more pronounced as the $ORDERED_LIST grows. Because GRAFT statements are very efficient, the use of $TEMP is preferable here.

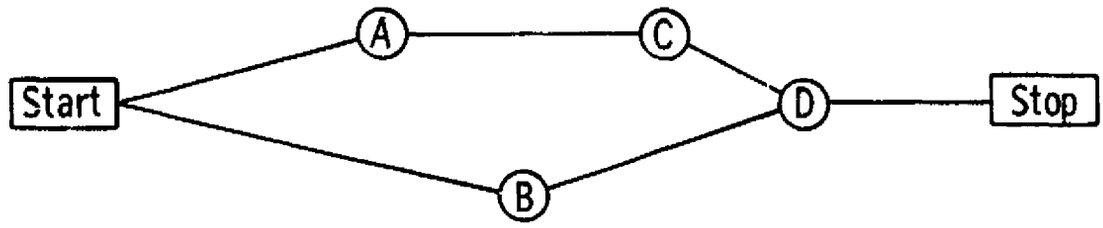
4.2 ELIMINATION OF REDUNDANT PREDECESSOR INFORMATION

4.2.1 Problem Statement

This example, like the previous one (Section 4.1), deals with a problem frequently encountered when working with job networks containing precedence relations. The program presented eliminates all redundant predecessor relations found in the input $JOBLIST. A predecessor is said to be redundant if it is not an immediate predecessor. For example, in Figure 4-2(a) jobs B and C are immediate predecessors of job D, while job A is a predecessor of a predecessor. In this case, if job A is shown in the list of predecessors of job D, it is redundant and should be removed from the list.

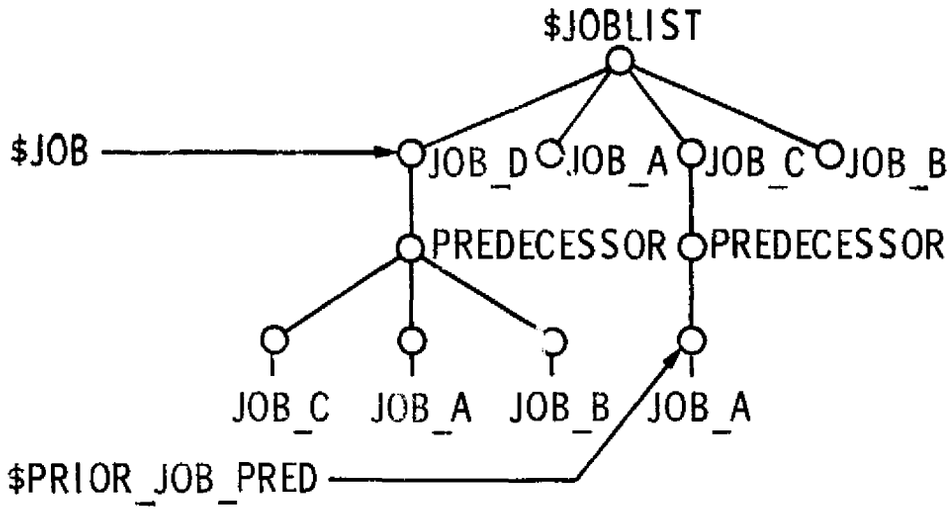
4.2.2 Program Logic

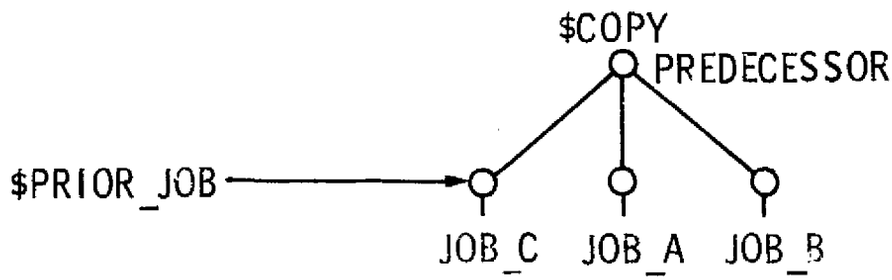
Considering the potential complexity of a large job network containing many redundantly specified predecessors, the solution program clearly requires some degree of sophistication. The PLANS program below, called REDUNDANT_PREDECESSOR_CHECKER, is quite short but powerful in the sense that it will handle the most general case. It takes advantage of some of the capabilities of PLANS (e.g., indirect referencing, tree pointers, "FIRST:", etc.) to efficiently eliminate all redundant predecessors in a single pass through $JOBLIST.



(a) Job Network Diagram

Tree Pointers





(b) Tree Pointers Used By REDUNDANT_PREDECESSOR_CHECKER

Fig 4-2 Sample Job Network Containing A Redundant Predecessor

The PLANS source code for REDUNDANT_PREDECESSOR_CHECKER is shown below.

```
 1 REDUNDANT_PREDECESSOR_CHECKER:  PROCEDURE($JOBLIST);
   /*  THIS PROCEDURE ELIMINATES ALL OF THE REDUNDANT PREDECESSORS      */
   /*  FOUND IN THE SET OF JOBS PASSED TO IT IN $JOBLIST.               */
 2 DECLARE $COPY,$JOB,$PRIOR_JOB,$PRIOR_JOB_PRED LOCAL;
 3 DO FOR ALL SUBNODES OF $JOBLIST USING $JOB;
 4    $COPY = $JOB.PREDECESSOR;
 5    DO FOR ALL SUBNODES OF $COPY USING $PRIOR_JOB;
 6       DO FOR ALL SUBNODES OF $JOBLIST.#(PRIOR_JOB).PREDECESSOR USING
             $PRIOR_JOB_PRED;
 7          IF $PRIOR_JOB_PRED ¬ELEMENT OF $COPY
 8             THEN $COPY(NEXT) = $PRIOR_JOB_PRED;
 9          PRUNE $JOB.PREDECESSOR(FIRST:$ELEMENT=$PRIOR_JOB_PRED);
10       END; /* $PRIOR_JOB_PRED */
11    END; /* $PRIOR_JOB */
12 END; /* $JOB */
13 END REDUNDANT_PREDECESSOR_CHECKER;
```

Three nested DO FOR ALL SUBNODES loops are used to scan $JOBLIST. Each DO loop specifies a tree pointer in its USING clause that facilitates iteration across the subnodes of the tree. The initial arrangement of the tree pointers for this data case is illustrated in Figure 4-2(b). Note that REDUNDANT_PREDECESSOR_CHECKER does not require that $JOBLIST be technologically ordered.

Briefly, the program logic operates as follows. The jobs are examined one at a time using $JOB (statement 3). First, the job's PREDECESSOR substructure is duplicated in $COPY (statement 4). Then, each predecessor's predecessor is checked to see if it appears in $COPY; if not, it is added there. This insures that eventually $COPY will contain all (both immediate and "indirect")

predecessors of $JOB. Since these nodes are added at $COPY(NEXT) they will be examined by the DO loop that iterates on the subnodes of $COPY. Statement 9 causes the redundant predecessor to be pruned. Note that if the "FIRST" such-that condition is not satisfied, indicating that a redundant predecessor was not found, no action will be taken by the PRUNE statement.

4.3 SELECTION OF A SPECIALIZED CREW

4.3.1 Problem Statement

Both of the previous examples dealt with generalized solutions to problems commonly encountered in precedence networks. This example presents custom-tailored logic to solve a specific problem. An English-language description of the "real world" problem is given below and then an example is given showing how the problem could be described and solved using PLANS.

Assume that a target of opportunity has been identified for a shuttle mission. A flight vehicle is available, but crewpersons must be trained for the special flight. There are six (6) candidate crewpersons, each with some, but not all, of the nine (9) required skill types. To fly the mission, a total of fifteen (15) "skill units" must be represented. These are defined in Table 1. A "skill unit" is the possession of a particular skill by a crewperson. Thus, if two crewpersons are required to be trained pilots, this represents two skill units. Training must be conducted as required to insure that the crewpersons who will fly have the required skills. The acquisition of the various skills requires training for the lengths of time shown in Table 2. No crewperson can participate in more than one training activity at any one time. Table 3 defines the skills that each candidate crewperson possesses initially. The mission can be flown using two, three, or four crewpersons as long as the flight team collectively satisfies the skill profile of Table 1. The objective is to select crewpersons whose training can be accomplished in minimum time and therefore permit the earliest launch.

Table 1 -- Required Skill Profile

Skill	Skill Distribution
Pilot	2
Observer	2
Paramedic	1
Camera Operator	2
Cook (knowledge of equipment)	2
Tape Recorder Expert	1
Mechanic	2
Electrician	2
Plumber	1
	15

Table 2 -- Training Times

Skill	Training Time
Pilot	16 days
Observer	4 days
Paramedic	2 days
Camera Operator	1 day
Cook (knowledge of equipment)	1 day
Tape Recorder Expert	9 days
Mechanic	12 days
Electrician	13 days
Plumber	8 days

Table 3 -- Available Crewpersons

Name	Initial Skills
John Doe	Pilot, Cook, Observer
Jack Smith	Pilot, Plumber
Ray Green	Tape
Jay Johnson	Pilot, Paramedic
Bob Schillings	Mechanic, Cook, Electrician
Mike Davis	Pilot, Mechanic, Cook

4.3.2 Problem Model

One of the first things that should be addressed by the programmer is how he might represent the three tables as PLANS tree structures. By separating this data from the source program, a great deal of program adaptability is gained. This is especially important if there is a high probability that the problem situation

will change slightly causing corresponding changes in one or more of the tables.

The tree structures used in this example are shown in Figure 4-3 exactly as they would appear in the input file to be read by the PLANS program. Note the easy one-to-one correspondence between tables and trees.

4.3.3 Program Logic

After drawing up a functional block diagram (high-level flowchart) the programmer will be ready to start writing the PLANS program. This can be easily accomplished by considering each block in the flowchart individually and then writing the PLANS statement(s) needed to perform its function. Although there are many possible ways of approaching this particular problem, the one implemented in Figures 4-4 and 4-5 was found to be quite straightforward. To demonstrate the close correspondence between the functional block diagram (Figure 4-4) and the block-structured PLANS program (Figure 4-5), the corresponding statement numbers are shown to the right of each block in Figure 4-4. It is recommended that serious readers examine the program in detail in order to understand how PLANS capabilities are used to solve this problem.

```
$SKILLS
   PILOT  -  2
   OBSERVER  -  2
   PARAMEDIC  -  1
   CAMERA  -  2
   COOK  -  2
   TAPE  -  1
   MECHANIC  -  2
   ELECTRICIAN  -  2
   PLUMBER  -  1
END
$TRAINING_TIMES
   PILOT  -  16
   OBSERVER  -  4
   PARAMEDIC  -  2
   CAMERA  -  1
   COOK  -  1
   TAPE  -  9
   MECHANIC  -  12
   ELECTRICIAN  -  13
   PLUMBER  -  8
END
$CREWMEN
   DOE
      SKILLS
         ¢  -  PILOT
         ¢  -  COOK
         ¢  -  OBSERVER
      AVAILABLE_DATE  -  1
   SMITH
      SKILLS
         ¢  -  PILOT
         ¢  -  PLUMBER
      AVAILABLE_DATE  -  1
   GREEN
      SKILLS
         ¢  -  TAPE
      AVAILABLE_DATE  -  1
   JOHNSON
      SKILLS
         ¢  -  PILOT
         ¢  -  PARAMEDIC
      AVAILABLE_DATE  -  1
   SCHILLINGS
      SKILLS
         ¢  -  MECHANIC
         ¢  -  COOK
         ¢  -  ELECTRICIAN
      AVAILABLE_DATE  -  1
   DAVIS
      SKILLS
         ¢  -  PILOT
         ¢  -  MECHANIC
         ¢  -  COOK
      AVAILABLE_DATE  -  1
END
```


Fig. 4-3 FIND_A_CREW Input Trees

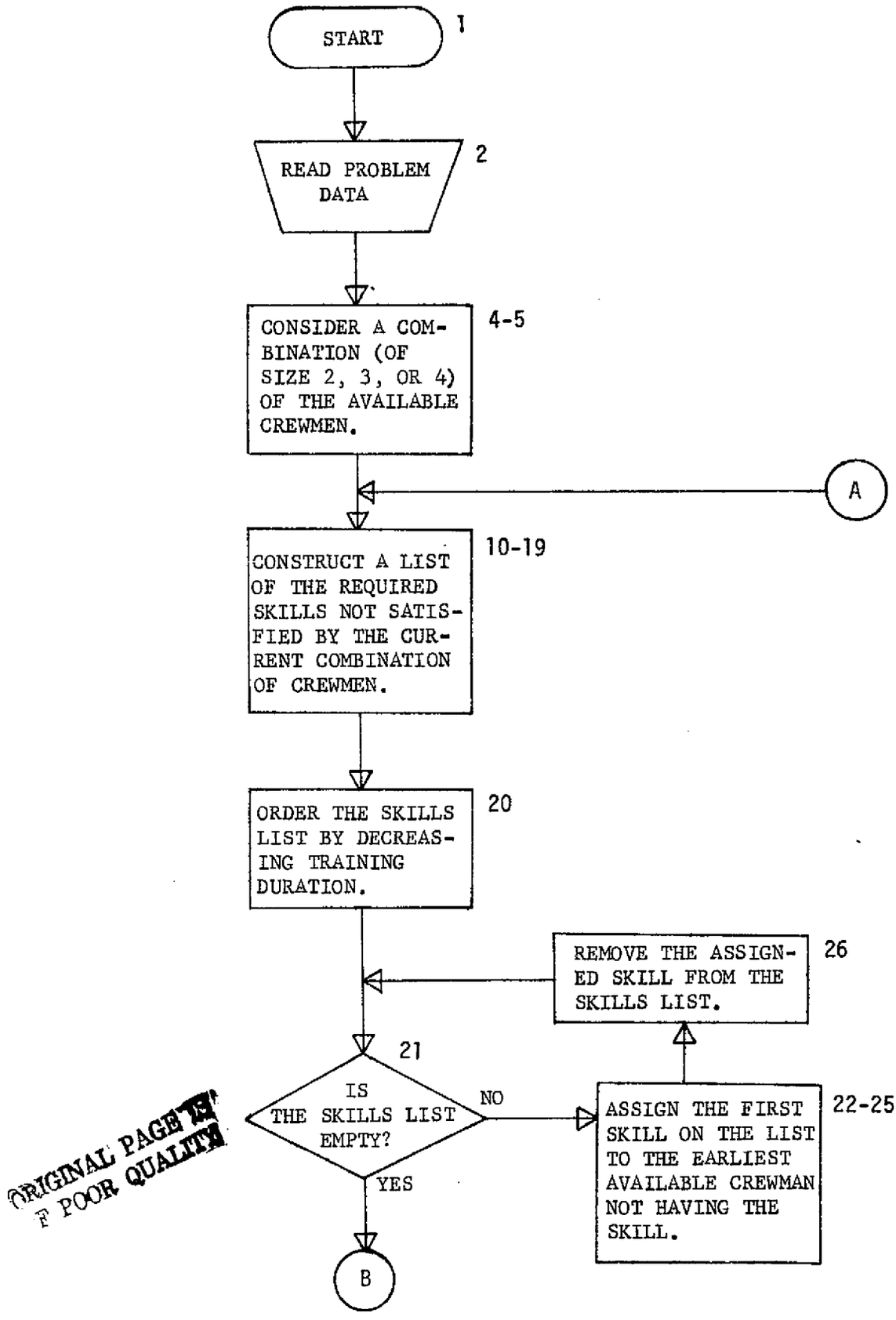


Fig. 4-4 FIND_A_CREW Functional Block Diagram

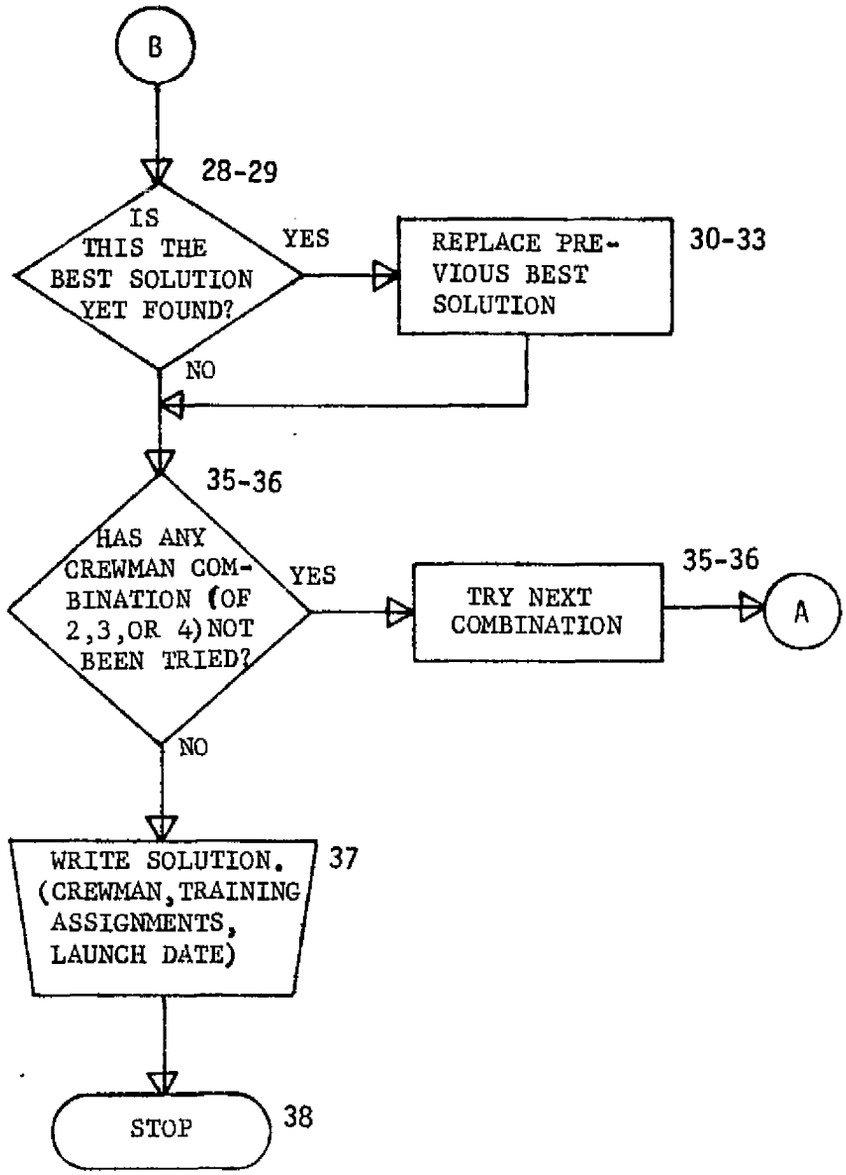

B
28-29
IS THIS THE BEST SOLUTION YET FOUND?
YES
REPLACE PREVIOUS BEST SOLUTION
30-33
NO
35-36
HAS ANY CREWMAN COMBINATION (OF 2,3,OR 4) NOT BEEN TRIED?
YES
TRY NEXT COMBINATION
35-36
A
NO
WRITE SOLUTION. (CREWMAN, TRAINING ASSIGNMENTS, LAUNCH DATE)
37
STOP
38

Fig. 4-4 (Concl.)

```
 1 FIND_A_CREW;  PROCEDURE;
   /*  THIS PROGRAM SELECTS A CREW FROM $CREWMEN WHICH HAS THE SKILLS    */
   /*  SPECIFIED IN $SKILLS AND WHICH REQUIRES THE LEAST AMOUNT OF       */
   /*  TRAINING TIME AS DETERMINED FROM $TRAINING_TIME.  THIS PERMITS    */
   /*  THE EARLIEST POSSIBLE LAUNCH.                                     */
 2       READ $SKILLS, $TRAINING_TIMES, $CREWMEN;
 3       BEST_LAUNCH = INFINITY;
 4       DO I = 4 TO 2 BY -1;
 5          DO FOR ALL COMBINATIONS OF $CREWMEN TAKEN I AT A TIME;
 6             PRUNE $CREW;
 7             DO J = 1 TO I;
 8                $CREW(J) = $COMBINATION(J);
 9                END;
10             DO FOR ALL SUBNODES OF $SKILLS USING $REQUIRED_NUMBER;
11                DO J = 1 TO $REQUIRED_NUMBER;
12                   INSERT $TRAINING_TIMES.#LABEL($REQUIRED_NUMBER)
                            BEFORE $DEFICIENCIES(FIRST);
13                   END;
14                END;
15             DO FOR ALL SUBNODES OF $CREW USING $PERSON;
16                DO FOR ALL SUBNODES OF $PERSON.SKILLS
                            USING $CURRENT_SKILL;
17                   PRUNE $DEFICIENCIES.#($CURRENT_SKILL);
18                   END;
19                END;
20             ORDER $DEFICIENCIES BY $ELEMENT;
21             DO WHILE ($DEFICIENCIES NOT IDENTICAL TO $NULL);
22                ORDER $CREW BY -AVAILABLE_DATE;
23                DEFINE $POINTER AS $CREW(FIRST:
                         LABEL($DEFICIENCIES(FIRST)) NOT ELEMENT OF
                         $ELEMENT.SKILLS);
24                $POINTER.SKILLS(NEXT) = LABEL($DEFICIENCIES(FIRST));
25                $POINTER.AVAILABLE_DATE = $POINTER.AVAILABLE_DATE +
                            $DEFICIENCIES(FIRST);
26                PRUNE $DEFICIENCIES(FIRST);
27                END;
28             ORDER $CREW BY AVAILABLE_DATE;
29             IF $CREW(FIRST).AVAILABLE_DATE < BEST_LAUNCH
30                THEN DO;
31                     $SOLUTION = $CREW;
32                     BEST_LAUNCH = $CREW(FIRST).AVAILABLE_DATE;
33                     END;
34             WRITE $CREW;
35             END;
36          END;
37       WRITE $SOLUTION, BEST_LAUNCH;
38       END FIND_A_CREW;
```

Fig. 4-5 FIND_A_CREW PLANS Program